Polygraphs provide a notion of presentation of an $n$-category:

an $n$-polygraph $P$ generates an $n$-category $P^*$
Polygraphs provide a notion of presentation of an $n$-category:

an $n$-polygraph $P$ generates an $n$-category $P^*$

*How can we describe $P^*$ in practice?*

(by “in practice” I mean a real implementation)
(in OCaml)
Polygraphs

A 0-polygraph

\[ \Sigma_0 \]

Example

0-polygraph

\[ x \quad y \]
Polygraphs

A 1-polygraph

Example

1-polygraph
A 1-polygraph generates a category

\[ \Sigma_0 \xrightarrow{s_0} \Sigma_1 \]
\[ \Sigma_0 \xleftarrow{s^* t_0} \Sigma^*_1 \]
\[ \Sigma_0 \xrightarrow{t^*_0} \Sigma_1 \]

**Example**

1-polygraph morphisms

\[ x \xrightarrow{a} y \]
\[ x \xrightarrow{a} y \xrightarrow{b} y \]
\[ x \xrightarrow{a} y \xrightarrow{b} y \xrightarrow{b} y \]
A 2-polygraph

\[
\begin{align*}
\Sigma_0 & \leq \Sigma_1^* \\
\Sigma_1 & \leq \Sigma_1^* \\
\Sigma_2 & \leq \Sigma_2^*
\end{align*}
\]

such that \( s_0^* \circ s_1 = s_0^* \circ t_1 \) and \( t_0^* \circ s_1 = t_0^* \circ t_1 \)

Example

1-polygraph morphisms

\[
\begin{align*}
x & \xrightarrow{a} y \\
x & \xrightarrow{b} y
\end{align*}
\]

2-polygraph

\[
\begin{align*}
x & \xrightarrow{a} y & b & \xrightarrow{b} y \\
x & \xrightarrow{a} y & b & \xrightarrow{b} y
\end{align*}
\]
A 2-polygraph generates a 2-category such that \( s_0^* \circ s_1 = s_0^* \circ t_1 \) and \( t_0^* \circ s_1 = t_0^* \circ t_1 \).

Example

1-polygraph morphisms

2-polygraph
Polygraphs

A 3-polygraph

\[ \Sigma_0 \xleftarrow{s_0} \Sigma_1 \xleftarrow{s_1^*t_0} \Sigma_1^* \xleftarrow{s_1^*t_1} \Sigma_2 \xleftarrow{s_2^*t_2} \Sigma_3 \]

such that \( s_1^* \circ s_2 = s_1^* \circ t_2 \) and \( t_1^* \circ s_2 = t_1^* \circ t_2 \)

Example

3-polygraph

\[ y \xrightarrow{b} y \]

\[ a \xrightarrow{\alpha} \Downarrow r \Downarrow \Downarrow s \]

\[ x \xrightarrow{a} y \xrightarrow{b} y \]

\[ x \xrightarrow{a} y \]
Implementing free categories generated by polygraphs

- I will present the ideas in order to construct the free categories they generates.
- This is largely based on Burroni’s notion of logograph.
- I will mainly focus on implementation issues.
- For simplicity, I will detail mostly the case of globular sets instead of general polygraphs.
- I want to convey the idea that when well-formulated, categorical constructions can be implemented “straightforwardly”.
Let’s start with a more simple example:

*graphs*
A graph is a diagram in Set:

$$\Sigma_0 \xleftarrow{s} \Sigma_1 \xrightarrow{t} \Sigma_0$$

with $\Sigma_0$ as objects and $\Sigma_1$ as vertices.

We write Graph for the category of graphs.
Implementing graphs

In OCaml (which we are using), a graph can be implemented as:

```ocaml
type vertex = unit ref

type graph =
  {
    vertices : vertex list;
    edges : (vertex * vertex) list;
  }
```
Constructing a graph

let x = ref ()
let y = ref ()
let f = (x,x)
let g = (x,x)
let h = (x,y)
let gr = { vertices = [x;y]; edges = [f;g;h] }
Constructing a graph

```ocaml
let x = ref ()
let y = ref ()
let f = (x,x)
let g = (x,x)
let h = (x,y)
let gr = { vertices = [x;y]; edges = [f;g;h] }
```

Notice that here we should be using *physical equality* in order to compare things:

```ocaml
# x = y;;
- : bool = true
# x == y;;
- : bool = false
# x == x;;
- : bool = true
```

```ocaml
# f = g;;
- : bool = true
# f == g;;
- : bool = false
# f == f;;
- : bool = true
```
Constructing a graph

```ocaml
text
let x = ref ()
let y = ref ()
let f = (x, x)
let g = (x, x)
let h = (x, y)
let gr = { vertices = [x; y]; edges = [f; g; h] }
```

Notice that here we should be using *physical equality* in order to compare things:

- we consider that our *universe* $\mathcal{U}$ is the collection of memory locations,
- constructions are invariant by action of the symmetric group on $\mathcal{U}$, i.e. what the garbage collector is doing.
Constructing a graph

``` Ocaml
let x = ref ()
let y = ref ()
let f = (x, x)
let g = (x, x)
let h = (x, y)
let gr = { vertices = [x; y]; edges = [f; g; h] }
```

Notice that here we should be using *physical equality* in order to compare things:

- it follows the general philosophy that a polygraph should describe a pure memory (pointer) data structure.
Constructing a graph

```ocaml
let x = ref ()
let y = ref ()
let f = (x,x)
let g = (x,x)
let h = (x,y)
let gr = { vertices = [x;y]; edges = [f;g;h] }
```

Notice that here we should be using \textit{physical equality} in order to compare things:

- we could also have assigned a different identifier (e.g. integer) to each vertex / edge, but this way of doing thing makes avoids many renaming issues.
Free graphs

The forgetful functor $U : \textbf{Cat} \to \textbf{Graph}$ admits a left adjoint: the free category on

$$G = \Sigma_0 \xleftarrow{s} \Sigma_1 \xrightarrow{t}$$

is the category $G^*$ with

- $\Sigma_0$ as objects
- $\Sigma_1^*$ as morphisms: the paths in the graph
Free categories

Given a graph

\[ G = \begin{array}{c}
  \xymatrix{x \ar[r]^{f} & x \\ & y \ar[l]_{g} \end{array} \]

a path (= morphism in the free category \( G^* \))

\[ \xymatrix{x \ar[r]^{f} & x \\ & x \ar[l]^{f} \\ & y \ar[u]_{g} \}
\]

can be seen as a labeled graph, i.e. a graph in the slice category

\textbf{Graph}/G

This is not the most immediate way of seeing things, but this point of view generalizes well!
Composing morphisms

Morphisms seen as labeled graphs can be composed by pushout:

\[
x \xrightarrow{f} x \xrightarrow{f} x \xrightarrow{g} y
\]

= \[
\begin{array}{c}
x \xrightarrow{f} x \xrightarrow{f} x \\
\end{array}
\]

\[
\begin{array}{c}
x \xrightarrow{g} y \\
\end{array}
\]

\[
\begin{array}{c}
X \\
\end{array}
\]
Paths

Of course, not every element of $\text{Graph}/G$ is a morphism in $G^*$:

\[
\begin{align*}
  x \xrightarrow{f} x & \quad \text{or} \quad x \rightarrow y \rightarrow x \\
  f & \quad \text{or} \quad f \swarrow \\

  x & \quad \text{or} \quad x \\
  f & \\
\end{align*}
\]

Moreover, two elements in $\text{Graph}/G$ represent the same morphism when they are isomorphic.
Paths

Of course, not every element of Graph/$G$ is a morphism in $G^*$:

\[
\begin{align*}
  x \xleftarrow{\sim} x & \quad \text{or} \quad x \quad y & \quad \text{or} \quad x \xrightarrow{f} x
\end{align*}
\]

Moreover, two elements in Graph/$G$ represent the same morphism when they are isomorphic.

The paths are precisely those which can be obtained by composing atomic graphs:

\[
\begin{align*}
  x \quad y & \quad x \xrightarrow{f} x & \quad x \xrightarrow{g} y
\end{align*}
\]

Of course, we can explicitly characterize “valid” paths here, but we won’t be able later on.
In this way, we can implement the free category over a graph.

The functions provided to the user (and not the data structure) will ensure that we can only construct valid paths.
Instead of giving details in this case, let’s go on with a slightly more elaborate (and more generic) example:

*globular sets*
An \textit{n-globular set} $G$ consists in

\begin{align*}
G_0 \xleftrightarrow{s_0}{t_0} G_1 \xleftrightarrow{s_1}{t_1} G_2 \xleftrightarrow{s_2}{t_2} \cdots \xleftrightarrow{s_{n-1}}{t_{n-1}} G_n
\end{align*}

such that

\begin{align*}
s_i \circ s_{i+1} &= s_i \circ t_{i+1} \\
t_i \circ s_{i+1} &= t_i \circ t_{i+1}
\end{align*}
An $n$-globular set $G$ consists in

\[
G_0 \xleftrightarrow{s_0} G_1 \xleftrightarrow{s_1} G_2 \xleftrightarrow{s_2} \cdots \xleftrightarrow{s_{n-1}} G_n
\]

\[
\xleftrightarrow{t_0} \xleftrightarrow{t_1} \xleftrightarrow{t_2} \cdots \xleftrightarrow{t_{n-1}}
\]

such that

\[
s_i \circ s_{i+1} = s_i \circ t_{i+1} \quad \text{and} \quad t_i \circ s_{i+1} = t_i \circ t_{i+1}
\]

and a morphism $f : G \to G'$ is

\[
G_0 \xleftrightarrow{s_0} G_1 \xleftrightarrow{s_1} G_2 \xleftrightarrow{s_2} \cdots \xleftrightarrow{s_{n-1}} G_n
\]

\[
\xleftrightarrow{t_0} \xleftrightarrow{t_1} \xleftrightarrow{t_2} \cdots \xleftrightarrow{t_{n-1}}
\]

\[
G_0' \xleftrightarrow{s_0'} G_1' \xleftrightarrow{s_1'} G_2' \xleftrightarrow{s_2'} \cdots \xleftrightarrow{s_{n-1}'} G_n'
\]

\[
\xleftrightarrow{t_0'} \xleftrightarrow{t_1'} \xleftrightarrow{t_2'} \cdots \xleftrightarrow{t_{n-1}'}
\]
An *n*-globular set $G$ consists in

$$
\begin{array}{cccccc}
G_0 & \xleftarrow{s_0} & G_1 & \xleftarrow{s_1} & G_2 & \xleftarrow{s_2} & \cdots & \xleftarrow{s_{n-1}} & G_n \\
& \xrightarrow{t_0} & & \xrightarrow{t_1} & & \xrightarrow{t_2} & & \xrightarrow{t_{n-1}} & \\
\end{array}
$$

such that

$$
\begin{align*}
  s_j \circ s_{i+1} &= s_j \circ t_{i+1} \\
  t_i \circ s_{i+1} &= t_i \circ t_{i+1}
\end{align*}
$$

and a morphism $f : G \to G'$ is

$$
\begin{array}{cccccc}
G_0 & \xleftarrow{s_0} & G_1 & \xleftarrow{s_1} & G_2 & \xleftarrow{s_2} & \cdots & \xleftarrow{s_{n-1}} & G_n \\
& \xrightarrow{t_0} & & \xrightarrow{t_1} & & \xrightarrow{t_2} & & \xrightarrow{t_{n-1}} & \\
\downarrow f_0 & & \downarrow f & & \downarrow f_2 & & \downarrow f & & \\
G'_0 & \xleftarrow{s'_0} & G'_1 & \xleftarrow{s'_1} & G'_2 & \xleftarrow{s'_2} & \cdots & \xleftarrow{s'_{n-1}} & G'_n \\
& \xrightarrow{t'_0} & & \xrightarrow{t'_1} & & \xrightarrow{t'_2} & & \xrightarrow{t'_{n-1}} & \\
\end{array}
$$

We write $\text{Glob}_n$ for their category.
The free category on a globular set

The same idea can be used in order to generate the free 
$n$-category over $n$-globular set $G$:

$n$-cells will be (some) elements of the slice category

\[ \text{Glob}_n/G \]

This is not the most simple way of implementing this (I would use Batanin’s trees for instance), but it generalizes to polygraphs.
The definition of the free $n$-category on a globular set $G$ is \textit{algebraic} (= free construction + relations), which we could use:

\begin{verbatim}
type morphism =
  | Generator of generator
  | Composition of morphism * int * morphism
  | Identity of morphism
\end{verbatim}

(a generator is an element of $G_i$). However, we would have to work modulo the relations of $n$-categories.

In the case of globular sets, there is a normal form for equivalence classes of terms modulo the relations such as exchange (this is what Batanin trees are), but there is no such thing for polygraphs.
Let’s implement things with the idea of morphisms in $\text{Glob}_n/G$. 
Generators

An element of $G_n$ will be:

type generator =

{
    dim : int;
    name : string;
    label : generator option;
    source : generator;
    target : generator;
}

where

- the name is only used for printing purposes,
- the label is here to be ready for the slice category.
Dummy generators

Since every generator has to have a source and a target, for elements of $G_0$ we will use:

```ml
let rec dummy_generator =
  {
    dim = -1;
    name = "dummy";
    label = None;
    source = dummy_generator;
    target = dummy_generator;
  }
```

and similarly for other inductive constructions.
Globular sets

Globular sets can then be implemented as:

```ocaml
type gset =
  {
    dim : int;
    generators : generator list;
    prev : gset;
  }
```

Again, notice that many things such as

```ocaml
assert (List.for_all (fun g -> g.dim = gset.dim) gset.generators);
```

will be maintained as an invariant.
Globular sets

Globular sets can then be implemented as:

```ocaml
type gset =
  {
    dim : int;
    generators : generator list;
    prev : gset;
  }
```

Again, notice that many things such as

```ocaml```
assert (List.for_all (fun g -> g.dim = gset.dim)
  gset.generators);
```

will be maintained as an invariants.
Some auxiliary functions are implemented. For instance, the canonical inclusion $G_n \hookrightarrow G_{n+1}$:

```ocaml
let degenerate gset =
  {
    dim = gset.dim + 1;
    generators = [];
    prev = gset;
  }
```

Moreover, I often use even-number functions for "clarity", e.g.

```ocaml
let create ~generators ~prev () =
  let dim = dim prev + 1 in
  assert (List.for_all (fun g -> G.dim g = dim) generators);
  { dim; generators; prev }
```
Some auxiliary functions are implemented. For instance, the canonical inclusion $G_n \hookrightarrow G_{n+1}$:

```ocaml
let degenerate gset =
  {
    dim = gset.dim + 1;
    generators = [];
    prev = gset;
  }
```

Moreover, I often use even dumber functions for “clarity”, e.g.

```ocaml
let create ~generators ~prev () =
  let dim = dim prev + 1 in
  assert (List.for_all (fun g -> G.dim g = dim) generators);
  { dim; generators; prev }
```
Morphisms between globular sets

Morphisms between globular sets are implemented as

type morphism =
{
  dim : int;
  source : gset;
  target : gset;
  map : (generator,generator) Mapq.t;
  prev : morphism;
}

where

- ('a,'b) Mapq.t implements a map (= set-theoretic function) from 'a to 'b where elements are compared with physical equality
Auxiliary functions

Many auxiliary functions can be easily implemented:

- **sequential composition:**
  \[
  \text{seq} : \text{morphism} \rightarrow \text{morphism} \rightarrow \text{morphism}
  \]

- **application of a morphism to a generator:**
  \[
  \text{app} : \text{morphism} \rightarrow \text{generator} \rightarrow \text{generator}
  \]

- **inclusion of a globular set into a bigger one:**
  \[
  \text{inclusion} : \text{gset} \rightarrow \text{gset} \rightarrow \text{morphism}
  \]

- **identity:**
  \[
  \text{id} : \text{gset} \rightarrow \text{morphism}
  \]

- etc.
Warning

Up to now, I have been simplifying a bit the code (not deeply, only to have clearer notations).

From now on, I just copy and paste, don’t hesitate to ask questions...
Non-disjoint union

The non-disjoint union of two globular sets:

```ocaml
define non_disjoint_union (s1 : globular_set) (s2 : globular_set) :=
  if dim s1 = dim s2 then
    let dim = dim s1 in
    match dim with
    | < 0 => dummy
    | _ => let prev = union (prev s1) (prev s2) in
      let generators =
        Listq.union (generators s1) (generators s2) in
      create ~generators ~prev ()
```
Relocating globular sets

Since we use physical equality, we can easily create an isomorphic copy of it:

```plaintext
define rec copy s =
define dim = dim s in
if dim < 0 then M.dummy else
  define f' = copy (prev s) in
define g_copy g =
  define source, target =
    if G.dim g = 0 then G.dummy, G.dummy
    else M.app f' (G.source g), M.app f' (G.target g)
in
  G.create ~name:(G.name g) ?label:(Option.find G.label g)
  ~source ~target ()
in
define map = Mapq.of_list (List.map (fun g -> g, g_copy g)
  (generators s)) in
define generators = List.map (Mapq.app map) (generators s) in
let target = create ~generators ~prev:(M.target f') () in
let f = M.create ~map ~prev:f' ~source:s ~target () in f
```
let coprod s1 s2 = 
  let i1 = copy s1 in
  let i2 = copy s2 in
  let s1' = M.target i1 in
  let s2' = M.target i2 in
  let s = union s1' s2' in
  let i1' = M.inclusion s1' s in
  let i2' = M.inclusion s2' s in
  M.seq i1 i1', M.seq i2 i2'
We can also have quotients:

- an equivalence class of elements $x \in X$ can be coded as set of pairs $(x, \hat{x})$,
- the canonical representative is chosen arbitrarily (e.g. the last inserted element)
- we can implement a “graded” version to have equivalence relations on globular sets, and compute a quotient globular set.
We can compute the coequalizer of two morphisms \( f_1 \) and \( f_2 \):

```ocaml
let coeq f1 f2 =
    assert (M.dim f1 = M.dim f2);
    assert (eq (M.source f1) (M.source f2));
    assert (eq (M.target f1) (M.target f2));
    let rec equiv f1 f2 =
        if M.dim f1 < 0 then Q.dummy else
            let r = equiv (M.prev f1) (M.prev f2) in
            let r = Q.degenerate ~set:(M.target f1) r in
            let r =
                List.fold_left (fun r g -> Q.add r (M.app f1 g) (M.app f2 g))
                    r (generators (M.source f1))
                in
                List.fold_left (fun r g -> Q.add r g g) r (generators (M.target f1))
            in
            let e = equiv f1 f2 in
            let s = M.target f1 in
            Q.set e
```
Pushouts

And thus pushouts (as we all know):

```ocaml
let pushout f1 f2 =
  assert (M.dim f1 = M.dim f2);
  assert (eq (M.source f1) (M.source f2));
  let i1, i2 = coprod (M.target f1) (M.target f2) in
  let g = coeq (M.seq f1 i1) (M.seq f2 i2) in
  M.seq i1 g, M.seq i2 g
```
With slightly more work, we can also compute the universal maps from a cocone of a coproduct or a pushout:
Now that we have all required operations on globular sets, we can implement free categories they generate.
type cell =
{
    dim : int; (** dimension *)
    set : gset; (** underlying globular set *)
    source : cell; (** source (n-1)-cell *)
    target : cell; (** target (n-1)-cell *)
}

where

- the underlying globular sets of the source and target are sub-globular sets of the underlying globular set of the cell.
From generators to cells

Any generator of dimension $n$ induces an $n$-cell with

- one (labeled) $n$-generator,
- two (labeled) $k$-generators for $0 \leq k < n$.

For instance, a 2-generator seen as a 2-cell is

\[
\begin{diagram}
\node{X} & & \node{y} \\
\node\(\downarrow\alpha\) & \node{\text{f}} & \node{\text{g}}
\end{diagram}
\]
Inductive definition

The 1-sphere

can be inductively obtained as a “bi-pushout”

and then we add a top dimensional generator labeled by $\alpha$. 
let of_generator s g =
  if G.dim g = 0 then
    let g' = G.create ~label:g
             ~source:G.dummy ~target:G.dummy ()
    in
    let set = S.add (S.degenerate S.dummy) g' in
    create ~set ~source:dummy ~target:dummy ()
  else
    ...

of_generator (1/3)
let rec sphere sgn s t =
  if G.dim s = 0 then ... else
  let c = sphere (S.prev sgn) (G.source s) (G.target t) in
  let gs = List.element (S.generators (set (source c))) in
  let gts = List.element (S.generators (set (target c))) in
  let gs = G.create ~label:s ~source:gs ~target:gts () in
  let gt = G.create ~label:t ~source:gs ~target:gts () in
  create ~set gs ~source gt ~target ()
let c = sphere (S.prev s) (G.source g) (G.target g) in
let source = List.element (S.generators (set (source c))) in
let target = List.element (S.generators (set (target c))) in
let g = G.create ~label:g ~name:(G.name g) ~source ~target ()
add c g
Isomorphic cells

Given two cells which are isomorphic, we can inductively construct the isomorphism between the underlying globular sets.

(there is only one isomorphism)

We call this function identify.
let seq d c1 c2 =
    assert (dim c1 = dim c2);
    assert (0 <= d && d < dim c1);
    let k = dim c1 - d in
    let t1 = iterate k target c1 in
    let s2 = iterate k source c2 in
    let f = identify t1 s2 in
    let f1, f2 = S.pushout (target_morphism ~k c1)
                 (M.seq (iterate k M.degenerate f)
                          (source_morphism ~k c2)) in
    let s = M.target f1 in
(* Recusive composition of the sources and targets. *)
let rec seq c1 f1 c2 f2 = ... in
let source, target =
    (if k = 1 then map (M.prev f1) (source c1)
     else seq (source c1) (M.prev f1) (source c2) (M.prev f2)),
    (if k = 1 then map (M.prev f2) (target c2)
     else seq (source c1) (M.prev f1) (source c2) (M.prev f2))
in
create ~set:s ~source ~target ()
Free $n$-categories on polygraphs is “the same”.

Notice that things get much more interleaved since we need the definition of the free $n$-category to define an $(n + 1)$-polygraph...

Let’s see the definition of polygraphs.
Polygraphs: generators

type generator =
{
    g_dim : int;
    g_name : string; (** name of the generator *)
    g_source : cell; (** source (n-1)-cell *)
    g_target : cell; (** target (n-1)-cell *)
}
and polygraph =
{
    p_dim : int;
    (** n-dimensional generators
        whose source and target are labeled
        in the underlying (n-1)-polygraph *)
    p_generators : generator list;
    (** underlying (n-1)-dimensional polygraph *)
    p_prev : polygraph;
}
and morphism =
{
  (** function between top-dimensional generators *)
  m_map : (generator,generator) Mapq.t;
  (** morphism between lower-dimensional cells *)
  m_prev : morphism;
  m_source : polygraph; (** source of the map *)
  m_target : polygraph; (** target of the map *)
}
and cell =
{
  (** labeling morphism for the polygraph of the cell *)
  c_label : morphism;
  c_source : cell; (** source (n-1)-cell *)
  (** inclusion of the polygraph of the source cell *)
  c_source_morphism : morphism;
  c_target : cell; (** target (n-1)-cell *)
  (** inclusion of the polygraph of the target cell *)
  c_target_morphism : morphism;
}
Most constructions can be performed similarly (excepted that everything is more complicated).
Face inclusions

Consider a 2-category with

- a 0-cell: $x$
- a 1-cell: $f : x \to x$
- 2-cells: $\eta : \text{id}_x \Rightarrow f$ and $\varepsilon : f \Rightarrow \text{id}_x$

The 2-cell $\eta$ is represented by the following polygraph:
Face inclusions

Consider a 2-category with

- a 0-cell: $x$
- a 1-cell: $f : x \to x$
- 2-cells: $\eta : \text{id}_x \Rightarrow f$ and $\varepsilon : f \Rightarrow \text{id}_x$

The 2-cell $\eta$ is represented by the following polygraph:
Consider a 2-category with:

- a 0-cell: $x$
- a 1-cell: $f : x \to x$
- 2-cells: $\eta : \text{id}_x \Rightarrow f$ and $\varepsilon : f \Rightarrow \text{id}_x$

The 2-cell $\eta$ is represented by the following polygraph:

\[
\begin{array}{c}
\text{X} \\
\downarrow f \\
\end{array}
\]
Consider a 2-category with
- a 0-cell: $x$
- a 1-cell: $f : x \to x$
- 2-cells: $\eta : \text{id}_x \Rightarrow f$ and $\varepsilon : f \Rightarrow \text{id}_x$

The 2-cell $\eta$ is represented by the following polygraph:

\[ \begin{array}{c}
X \\
\downarrow \ \\
\downarrow \\
\downarrow \\
f \\
\end{array} \]
Consider a 2-category with

- a 0-cell: $x$
- a 1-cell: $f : x \to x$
- 2-cells: $\eta : \text{id}_x \Rightarrow f$ and $\varepsilon : f \Rightarrow \text{id}_x$

The 2-cell $\eta$ is represented by the following polygraph:

The source is not a sub-polygraph: we have to have an explicit inclusion.
let uid = uidebug in
let p0 = P.degenerate P.dummy in
let star = G.create ~name:"*" ~source:C.dummy ~target:C.dummy () in
let p0 = P.add p0 star in
let star_c = C.of_generator p0 star in
let p1 = P.degenerate p0 in
let one = G.create ~name:"a" ~source:star_c ~target:star_c () in
let p1 = P.add p1 one in
let one_c = C.of_generator p1 one in
let p2 = P.degenerate p1 in
let two_c = C.seq 0 one_c one_c in
let mu = G.create ~name:"µ" ~source:(C.seq 0 one_c one_c) ~target:one_c () in
let p2 = P.add p2 mu in
let mu_c = C.of_generator p2 mu in
let mumumu = C.seq 1 (C.seq 0 mu_c mu_c) mu_c in
Printf.printf "mumumu:\n%s\n%!" (C.to_string uid mumumu)
An example

mumumu:

2-cell:

dim 0:
  *39(*0) *46(*0) *40(*0) *38(*0) *43(*0)

dim 1:
  a37(a0) : *1(*46) → *1(*39)
  a39(a0) : *1(*40) → *1(*46)
  a41(a0) : *1(*40) → *1(*39)
  a33(a0) : *1(*43) → *1(*40)
  a35(a0) : *1(*38) → *1(*43)
  a42(a0) : *1(*38) → *1(*40)
  a32(a0) : *1(*38) → *1(*39)

dim 2:
  μ6(μ0) : *9(*40) -a4(a39)→ *7(*46) -a5(a37)→ *8(*39)
    => *3(*40) -a1(a41)→ *2(*39)
  μ7(μ0) : *9(*38) -a4(a35)→ *7(*43) -a5(a33)→ *8(*40)
    => *3(*38) -a1(a42)→ *2(*40)
  μ8(μ0) : *9(*38) -a4(a42)→ *7(*40) -a5(a41)→ *8(*39)
    => *3(*38) -a1(a32)→ *2(*39)
Compositions

As before, composition of $f$ and $g$ can be performed given an isomorphism

$$\partial^+(f) \cong \partial^-(g)$$

However:

- there is no necessarily a unique isomorphism anymore, e.g. along “floating 2-cells” (see the blackboard),
- I expect that this could be fixed by using the “algebraic construction” on top of polygraphs in order to keep track of the cells?
What’s next

- We have an implementation of polygraphs.
- I could not (yet) implement more interesting operations such as homotopic reduction: we can fake \((n, k)\)-polygraphs, but the real issue is how to perform substitution of cells.
- We should implement other meaningful operations: pattern matching (so that we can rewrite), Tietze transformations, homotopic reduction, termination (Guiraud’s derivations), etc.
- We need to interface traditional tools (e.g. rewriting presentations of monoids).