IMPLEMENTING POLYGRAPHS

SAMUEL MIMRAM

CATHRE meeting

February 2nd 2015

Polygraphs

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

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

Polygraphs

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

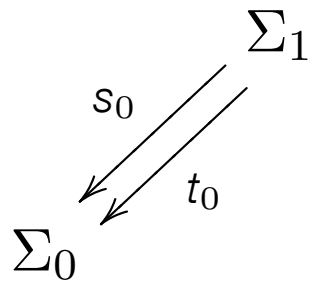
Σ_0

Example

0-polygraph

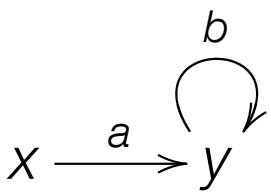
$x \qquad y$

A 1-polygraph



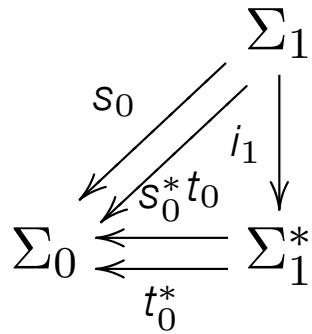
Example

1-polygraph



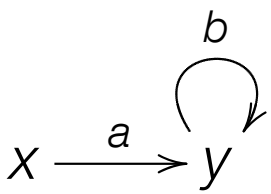
Polygraphs

A 1-polygraph generates a category

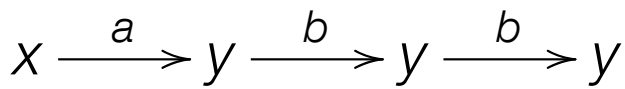


Example

1-polygraph

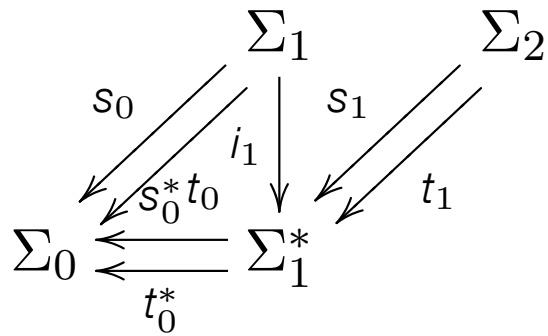


morphisms



Polygraphs

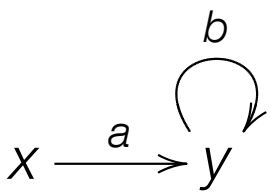
A 2-polygraph



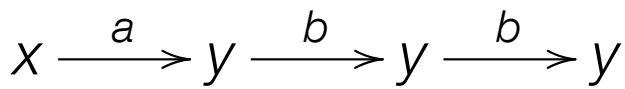
such that $s_0^* \circ s_1 = s_0^* \circ t_1$ and $t_0^* \circ s_1 = t_0^* \circ t_1$

Example

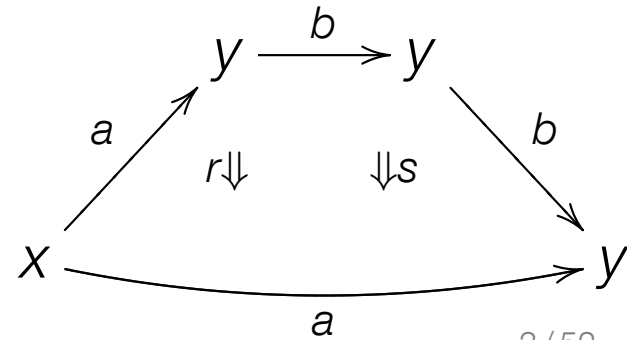
1-polygraph



morphisms

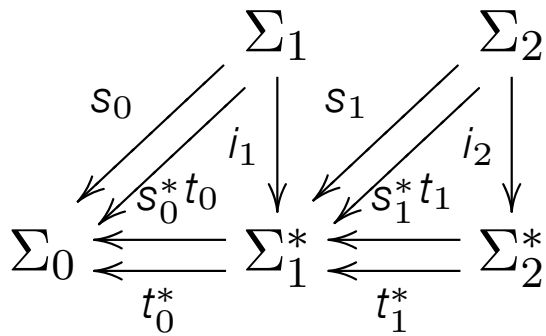


2-polygraph



Polygraphs

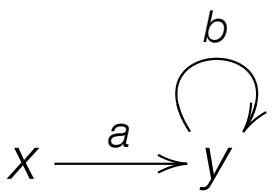
A 2-polygraph generates a 2-category



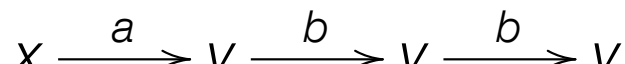
such that $s_0^* \circ s_1 = s_0^* \circ t_1$ and $t_0^* \circ s_1 = t_0^* \circ t_1$

Example

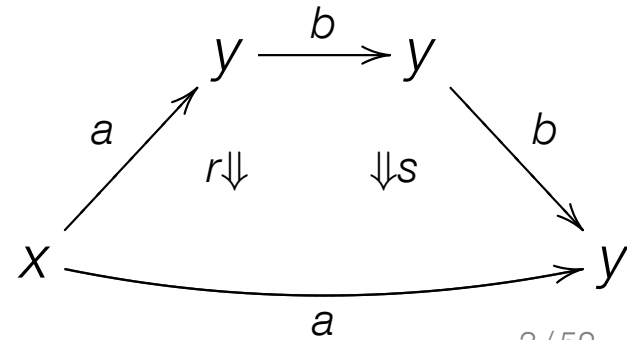
1-polygraph



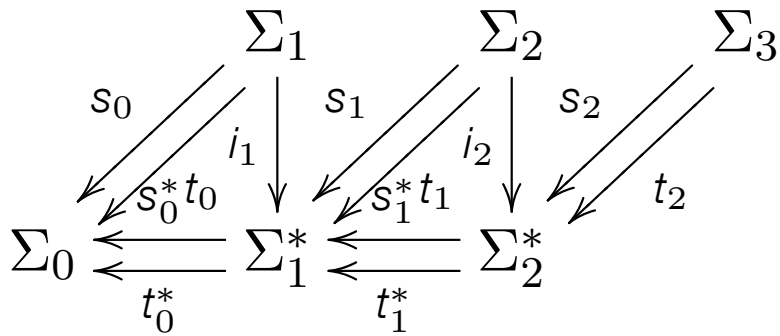
morphisms



2-polygraph



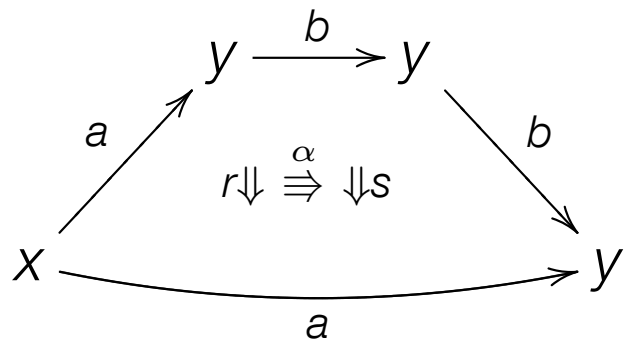
A 3-polygraph



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



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

Graphs

A **graph** is a diagram in **Set**:

$$\Sigma_0 \underset{t}{\overset{s}{\leftleftarrows}} \Sigma_1$$

with Σ_0 as objects and Σ_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:

```
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

```
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:

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

```
# f = g;;
- : bool = true
# f == g;;
- : bool = false
# f == f;;
- : bool = true
```

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] }
```

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

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

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] }
```

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

```
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 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 : **Cat** $\to$ **Graph** admits a *left adjoint*: the free category on

$$G \qquad = \qquad \Sigma_0 \underset{t}{\overset{s}{\leftleftarrows}} \Sigma_1$$

is the category G^* with

- Σ_0 as objects
- Σ_1^* as morphisms: the paths in the graph

Free categories

Given a graph

$$G \quad = \quad \overset{f}{\circlearrowright} \; x \xrightarrow{g} y$$

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

$$x \xrightarrow{f} x \xrightarrow{f} x \xrightarrow{g} y$$

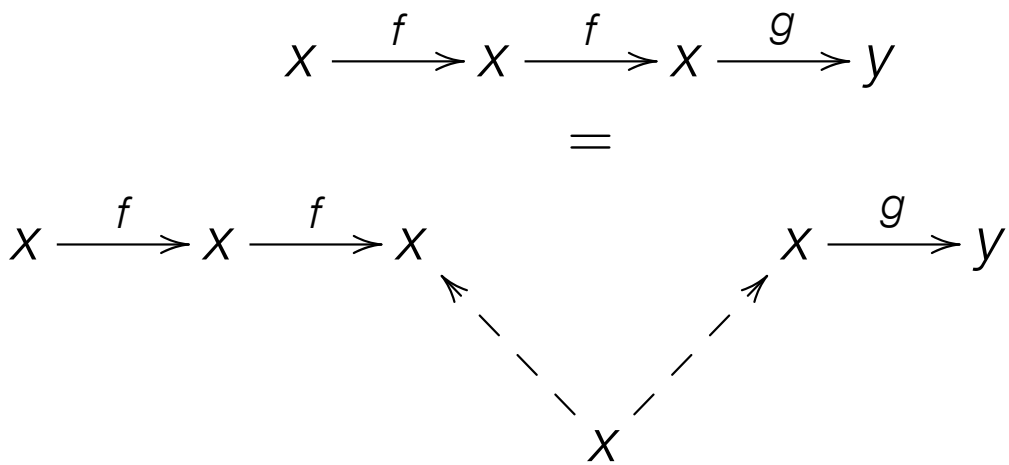
can be seen as a *labeled graph*, i.e. a graph in the slice category

$$\mathbf{Graph}/G$$

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

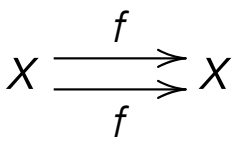
Composing morphisms

Morphisms seen as labeled graphs can be composed by pushout:



Paths

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



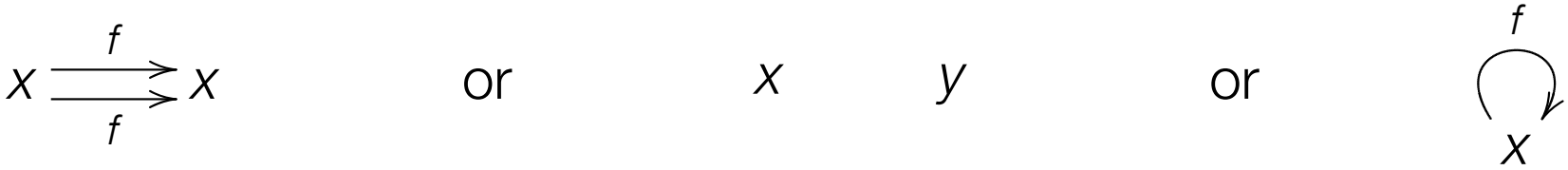
or

or

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

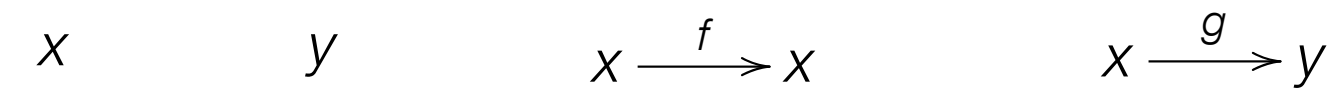
Paths

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



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*:



Of course, we can explicitly characterize “valid” paths here, but we won’t be able later on.

Invariants

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

Globular sets

An n-**globular set** G consists in

$$G_0 \underset{t_0}{\overset{s_0}{\leftleftarrows}} G_1 \underset{t_1}{\overset{s_1}{\leftleftarrows}} G_2 \underset{t_2}{\overset{s_2}{\leftleftarrows}} \quad \cdots \quad \underset{t_{n-1}}{\overset{s_{n-1}}{\leftleftarrows}} G_n$$

such that

$$s_i \circ s_{i+1} = s_i \circ t_{i+1} \qquad t_i \circ s_{i+1} = t_i \circ t_{i+1}$$

Globular sets

An n-**globular set** G consists in

$$G_0 \underset{t_0}{\overset{s_0}{\leftleftarrows}} G_1 \underset{t_1}{\overset{s_1}{\leftleftarrows}} G_2 \underset{t_2}{\overset{s_2}{\leftleftarrows}} \quad \cdots \quad \underset{t_{n-1}}{\overset{s_{n-1}}{\leftleftarrows}} G_n$$

such that

$$s_i \circ s_{i+1} = s_i \circ t_{i+1} \qquad t_i \circ s_{i+1} = t_i \circ t_{i+1}$$

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

$$\begin{array}{ccccccccc}
G_0 & \underset{t_0}{\overset{s_0}{\leftleftarrows}} & G_1 & \underset{t_1}{\overset{s_1}{\leftleftarrows}} & G_2 & \underset{t_2}{\overset{s_2}{\leftleftarrows}} & \cdots & \underset{t_{n-1}}{\overset{s_{n-1}}{\leftleftarrows}} & G_n \\
\downarrow{\scriptstyle f_0} & & \downarrow{\scriptstyle f_f} & & \downarrow{\scriptstyle f_2} & & & & \downarrow{\scriptstyle f_n} \\
G'_0 & \underset{t'_0}{\overset{s'_0}{\leftleftarrows}} & G'_1 & \underset{t'_1}{\overset{s'_1}{\leftleftarrows}} & G_2 & \underset{t'_2}{\overset{s'_2}{\leftleftarrows}} & \cdots & \underset{t'_{n-1}}{\overset{s'_{n-1}}{\leftleftarrows}} & G'_n
\end{array}$$

Globular sets

An n-**globular set** G consists in

$$G_0 \underset{t_0}{\overset{s_0}{\leftleftarrows}} G_1 \underset{t_1}{\overset{s_1}{\leftleftarrows}} G_2 \underset{t_2}{\overset{s_2}{\leftleftarrows}} \quad \cdots \quad \underset{t_{n-1}}{\overset{s_{n-1}}{\leftleftarrows}} G_n$$

such that

$$s_i \circ s_{i+1} = s_i \circ t_{i+1} \qquad t_i \circ s_{i+1} = t_i \circ t_{i+1}$$

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

$$\begin{array}{ccccccc}
G_0 & \underset{t_0}{\overset{s_0}{\leftleftarrows}} & G_1 & \underset{t_1}{\overset{s_1}{\leftleftarrows}} & G_2 & \underset{t_2}{\overset{s_2}{\leftleftarrows}} & \cdots \quad \underset{t_{n-1}}{\overset{s_{n-1}}{\leftleftarrows}} \quad G_n \\
\downarrow{\scriptstyle f_0} & & \downarrow{\scriptstyle f_f} & & \downarrow{\scriptstyle f_2} & & \downarrow{\scriptstyle f_n} \\
G'_0 & \underset{t'_0}{\overset{s'_0}{\leftleftarrows}} & G'_1 & \underset{t'_1}{\overset{s'_1}{\leftleftarrows}} & G_2 & \underset{t'_2}{\overset{s'_2}{\leftleftarrows}} & \cdots \quad \underset{t'_{n-1}}{\overset{s'_{n-1}}{\leftleftarrows}} \quad G'_n
\end{array}$$

We write $\mathbf{Glob}_n$ 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

$$\mathbf{Glob}_n/G$$

This is not the most simple way of implementing this (I would use Batanin's trees for instance), but it generalizes to polygraphs.

An algebraic approach

The definition of the free n-category on a globular set G is *algebraic* (= free construction + relations), which we could use:

```
type morphism =
  | Generator of generator
  | Composition of morphism * int * morphism
  | Identity of morphism
```

(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 $\mathbf{Glob}_n/G$.

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:

```
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:

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

Globular sets

Globular sets can then be implemented as:

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

Again, notice that many things such as

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

will be maintained as an invariants.

Auxiliary functions

Some auxiliary functions are implemented. For instance, the canonical inclusion $G_n \hookrightarrow G_{n+1}$:

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

Auxiliary functions

Some auxiliary functions are implemented. For instance, the canonical inclusion $G_n \hookrightarrow G_{n+1}$:

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

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

```
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:
  `seq : morphism -> morphism -> morphism`
- application of a morphism to a generator:
  `app : morphism -> generator -> generator`
- inclusion of a globular set into a bigger one:
  `inclusion : gset -> gset -> morphism`
- identity:
  `id : gset -> 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:

```
let rec union s1 s2 =
  assert (dim s1 = dim s2);
  let dim = dim s1 in
  if dim < 0 then dummy else
    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:

```
let rec copy s =
  let dim = dim s in
  if dim < 0 then M.dummy else
    let f' = copy (prev s) in
    let g_copy g =
      let 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
    let map = Mapq.of_list (List.map (fun g -> g, g_copy g)
                (generators s)) in
   let 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
```

Coproduct

```
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'
```

Quotients

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.

Coequalizers

We can compute the coequalizer of two morphisms `f1` and `f2`:

```
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
  in
  let e = equiv f1 f2 in
  let s = M.target f1 in
  Q.set e
```

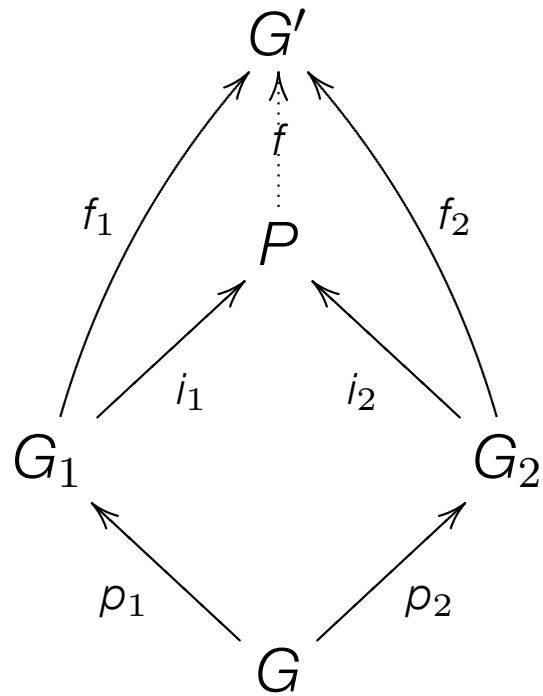
Pushouts

And thus pushouts (as we all know):

```
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
```

Universal maps

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.

Cells

```
type cell =
    {
      dim : int; (** dimension *)
      set : gset; (** underlying globular set *)
      source : cell; (** source (n-1)-cell *)
      target : cell; (** target (n-1)-cell *)
    }
```

where

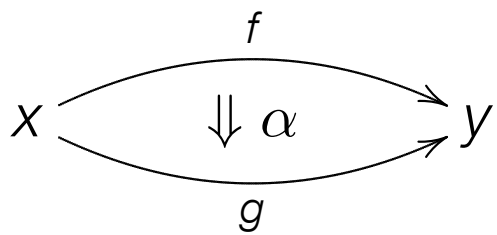
- the underlying globular sets of the source and target are sub-globular sets of the underlying globular set of the cell.

From generators to cells

Any generator of dimension n induces an n-cell with

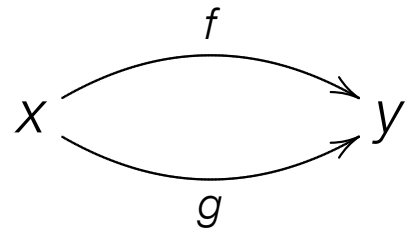
- one (labeled) n-generator,
- two (labeled) k-generators for $0 \leq k < n$.

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

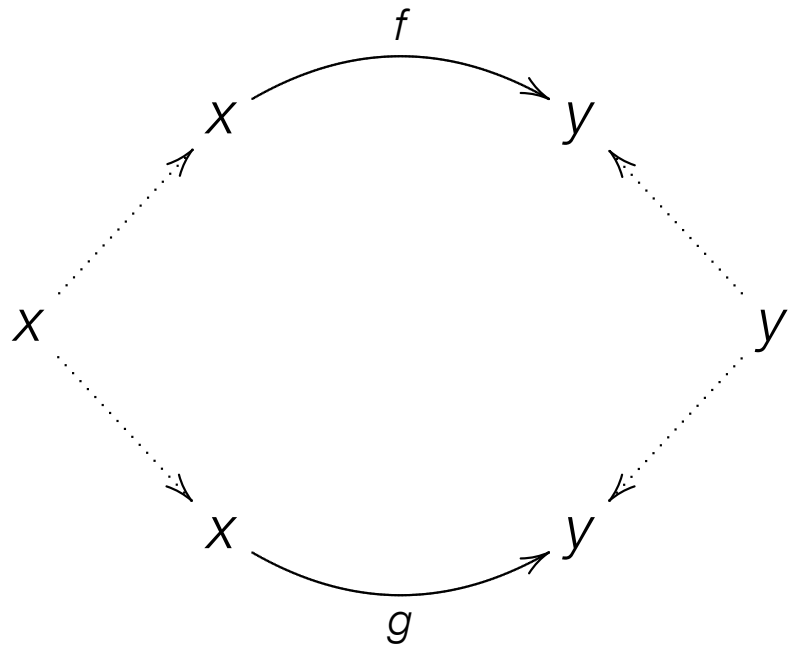


Inductive definition

The 1-sphere



can be inductively obtained as a “bi-pushout”



and then we add a top dimensional generator labeled by α.

of_generator (1/3)

```
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 (2/3)

```
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 gsrc' = List.element (S.generators (set (source c))) in
   let gtgt' = List.element (S.generators (set (target c))) in
   let gsrc = G.create ~label:s ~source:gsrc' ~target:gtgt' (
   let gtgt = G.create ~label:t ~source:gsrc' ~target:gtgt' (
    let source = add c gsrc in
    let target = add c gtgt in
    let set = set c in
    let set = S.add set gsrc in
    let set = S.add set gtgt in
    let set = S.degenerate set in
    create ~set ~source ~target ()
in
...
```

of_generator (3/3)

```
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`.

Composition

```
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 *)
    }
```

Polygraphs: polygraphs

```
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;
  }
```

Polygraphs: morphisms

```
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 *)
  }
```

Polygraphs: cells

```
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: $\eta : \mathrm{id}_x \Rightarrow f$ and $\varepsilon : f \Rightarrow \mathrm{id}_x$

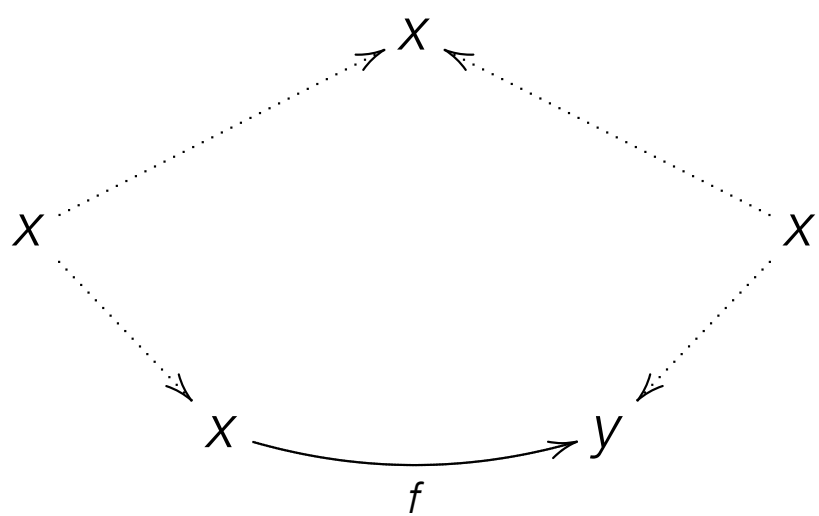
The 2-cell η is represented by the following polygraph:

Face inclusions

Consider a 2-category with

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

The 2-cell η 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: $\eta : \mathrm{id}_x \Rightarrow f$ and $\varepsilon : f \Rightarrow \mathrm{id}_x$

The 2-cell η 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: $\eta : \mathrm{id}_x \Rightarrow f$ and $\varepsilon : f \Rightarrow \mathrm{id}_x$

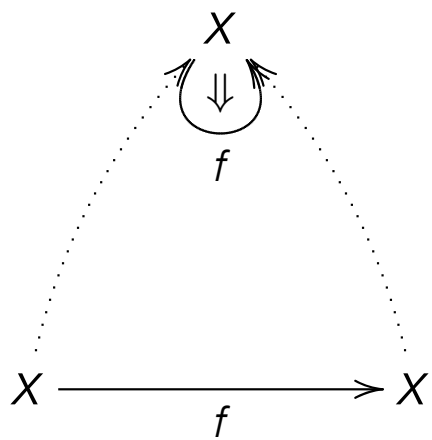
The 2-cell η is represented by the following polygraph:

Face inclusions

Consider a 2-category with

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

The 2-cell η is represented by the following polygraph:



The source is not a sub-polygraph: we have to have an explicit inclusion.

An example

```
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:o
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

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

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).