University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Jacek Karwowski
Student no. 359713

Products in positive opetopic sets

Bachelor's thesis
in MATHEMATICS

Supervisor:
dr. hab. Marek Zawadowski
Institute of Mathematics

August 2018

Supervisor's statement

Hereby I confirm that the presented thesis was prepared under my supervision and that it fulfils the requirements for the degree of Bachelor of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the presented thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Abstract

Opetopes are geometric objects, similar to simplices, having - on one hand, still simple, but on the other - much more complicated than above, structure. They can be used to encode the structure of composition, having applications in category theory.
Our main results are: first, algorithm for construction of product of two opetopes, together with two different implementations; second, a proof that this algorithm is correct - in particular, that it always terminates - thus proving the fact that the product of two opetopes is a finite object [1].
Our main motivation, although being quite far reached, is the potential usage of opetopes in construction of a new model of homotopy type theory. This requires a deeper understanding of the category of opetopes - does it behave in a "sane" way, preserving geometric intuitions (for example - does a product of two opetopes is contractible)? We hope that results presented here are steps on the way to answering this problem.

Keywords

opetope, category theory, homotopy type theory

Thesis domain (Socrates-Erasmus subject area codes)

11.1 Matematyka

Subject classification

18 Category theory; homological algebra
18F20 Presheaves and sheaves

Tytuł pracy w języku polskim

Produkty w pozytywnych zbiorach opetopowych

[1] for some specific meaning of *finite*

Abstract

Opetopy to obiekty geometryczne, podobne do sympleksów, mające z jednej strony jeszcze nadal prostą, ale z drugiej - dużo bardziej od powyższych, strukturę. Jednym z ich właściwości jest kodowanie struktury złożenia funkcji, więc znajdują zastosowania głównie w teorii kategorii.
Nasze główne rezultaty, to: po pierwsze, algorytm konstrukcji produktu dwóch opetopów, razem z jego dwoma implementacjami; po drugie, dowód, że powyższy algorytm jest poprawny, czyli, między innymi, że kończy się w skończonym czasie na każdym poprawnym wejściu - co jest bezpośrednim dowodem, że produkt dwóch opetopów jest obiektem skończonym [2].
Główna (chociaż dość odległa z technicznego punktu widzenia) motywacja stojąca za tą pracą to potencjalna możliwość wykorzystania zbiorów opetopwych do konstrukcji nowego modelu homotopijnej teorii typów. Mamy nadzieję, że rezultaty zaprezentowane w pracy będą użyteczne na drodze do tego celu.

[2] w pewnym sensie słowa *skończony*

Contents

Introduction

Opetopes are not very widely known mathematical objects - this is an indisputable fact. I myself learned about them only at the beginning of writing this dissertation, not having them mentioned even once during my three-year journey through BSc in Mathematics.
I stumbled across them because I wanted to understand homotopy type theory ([1]) - a new mathematical framework, lying on the intersection of theoretical computer science, type theory & programming languages theory, and mathematics of topology, category theory and logic. I was told by a professor that since HoTT is so young, there are yet not many models of it - and the ones that currently exist are not satisfactory in many ways. Opetopes, being more sophisticated objects than simplicial, or cubical, sets, could provide a way of developing new models for HoTT, thereby expanding our understanding of the theory. But first, there are questions that have to be answered, questions quite apart from the main framework of this new theory.
So, this work was born from this very need. Understanding category of opetopes and opetopic sets is fundamental for further progress - and one of the simplest categorical constructions is a product. We hereby focused on a general way of computing such products, and present it here along with auxiliary definitions and lemmas.
So, even if opetopes are not widely known, they are certainly very interesting objects in themselves (and there is much more about them than it is described here - see, for example, footnote in the third chapter). We invite reader to take interest in them.

Structure of this work is, as follows: First chapter serves as an introduction. It presents a process of generalization from a geometric definition of a simplex, ending in a quite abstract categorical definition. This is a material that is widely know, it can be freely skipped by someone already familiar with simplicial sets.
Exposition of topics in first chapter roughly follows [2]. I've drastically shortened up the presentation and left only the information that is needed to build intuition for later chapters, but any interested reader should definitely consult the source material for a much more detailed explanation.
Second chapter consists of definition of opetopes, along with some intuition on how to think about them. Again, material covered here is taken from a source - [3] and [4], but it probably shouldn't be skipped. This is because there are some technical differences in literature concerning definition of opetopes - in particular, on what are the morphisms between them.
Third and the main chapter is the main content of this dissertation. It begins with the description of the construction of a product $P \times Q$, for opetopes P and Q. We then prove two main lemmas - that the algorithm outputs all faces, and that it finishes in a finite time. The first lemma is actually the most complicated part of this work, since there is some technical buildup required.
Fourth chapter discusses the coded part of this work. In particular, it introduces reader to the two implementations and presents a short summary of (computational) results. It does

not introduce any mathematical concepts, but is probably worth reading just for the intuition on how quickly products of opetopes become intractably huge.
There are also two appendices, consisting of the main parts of the code written for this dissertation. These are just the main parts, required to possibly replicate experiments. Full code, in particular parts dealing with testing, are available online, with links provided in the implementation chapter.

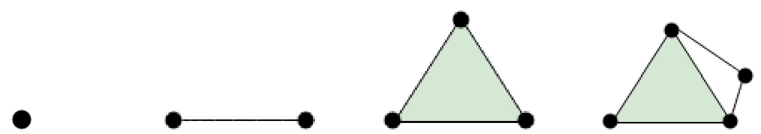
Acknowledgements I would like to express my gratitude for my supervisor, dr Marek Zawadowski. Overwhelming part of this dissertation bases on his research - in particular the main concepts for the algorithm of construction and proofs were developed in collaboration with him. His kind support and patience have helped me tremedously in understanding the material presented here.

Chapter 1

Simplices, symplicial complexes and simplicial sets

As simplicial sets are essentially a simpler version of opetopic sets (as any simplicial set can be modeled as appropriate opetopic set), understanding them first is probably a good way of starting with opetopes. We will start with most natural (and most popular) definition of simplex and then present a way to generalize, up to categorical definition of simplicial sets.

Simplices are basically a higher-dimensional generalizations of a triangle.



(a) 0-simplex (b) 1-simplex (c) 2-simplex (d) 3-simplex

Definition 1.0.1. A set $\Delta = \{(x_1, ..., x_n) \in \mathbb{R}^{n+1}\}$ is an n-dimensional (geometric) simplex, with a set of vertices $\delta = \{p_1, ...p_{n+1}\}$, if the set δ is affine independent, and δ is a convex hull over Δ. A simplex $\Gamma \subset \Delta$ whose set of vertices is a subset of δ is called a face of Δ.

Definition 1.0.2. Simplex Δ of dimension n is called standard simplex, if its set of vertices is given by the standard basis of $\mathbb{R}^{n+1}$.

Simplicial complexes are objects we can build by gluing together simplices.

Definition 1.0.3. A set of simplices $K = \{K_1, ..., K_n\}$ is a (geometric) simplicial complex, if for all K_i, any face of K_i is in K, and if intersection of any K_i, K_j is either empty or is their common face.

These definitions are close to the geometric intuition, but they are quite complicated - for example, they require a definition of $\mathbb{R}^n$ space. If we are to look at objects from completely topological perspective, it doesn't matter how they are embedded in the space - all that interests us is the homeomorphism type. So, we should look for a simpler, more economical representation.

Definition 1.0.4. An (abstract) simplicial complex is a set X^0 of vertices, together with a family of sets X_k for all $k \in \mathbb{N}$. Each X^k is a set of $k+1$-element subsets of X^0, satisfying the condition that for every element of X^k, its every $j+1$-element subset belongs to X^j.

This way, we lose all information about embedding of any particular simplicial complex, but we preserve the information required to reconstruct it (up to the homeomorphism) - we can get back geometric simplex simply by mapping elements of X^k to standard simplicial complexes, and then gluing them together using quotient topology.

A map between (geometrical) simplicial complexes is just continuous function between them. A map between abstract simplicial complexes is determined by its values on vertices - it can be realized as map between geometric complexes by linear interpolation.

Another step in generalization is adding a requirement that there is total order on elements X^0. This doesn't change anything with respect to the previously defined representation by sets of appropriate cardinality - we now have simply a "standard" way of representing a simplex (e.g. $[v_{i_0}, ..., v_{i_{n+1}}]$ is a representation of a face $\{v_{i_0}, ..., v_{i_{n+1}}\} \in X^n$ if $v_{i_k} < v_{i_l}$ if only $i_k < i_l$). Using this fact, we can define maps of simplicial complexes as order-preserving functions. In fact, we can now think of the complex X as the collection of inclusion maps of the simplices that make up X. This fact will be formalized in a moment, in categorical definition of a simplicial complex.

Now, having an $n+1$-dimensional ordered simplex X, we would like to have a way of referring to its n-dimensional faces. We thus define a collection of *face maps* $d_1, ..., d_{n+1}$, such that $d_j[0, ..., n+1] = [0, ..., j-1, j+1, ..., n+1]$. Extending this definition, we define a similar collection of maps on any (ordered) simplicial complex.
Abusing the notation slightly, we will not write d_i^n (n selecting dimension), instead using d_i for any of theses maps.
A simple argument shows that any face map going by more than one dimension (e.g. $X^{l+k} \to X^l$) can be uniquely decomposed into $d_{i_1}, ..., d_{i_k}$ maps, with the restriction that $i_j < i_{j+1}$ for all j.

In addition to face maps, we can also talk about *degeneracy maps*. As face maps go from simplex to its face, a degeneracy map, in a way, embeds higher-dimensional simplex in a lower-dimensional one. For a $n+1$ dimensional face, we define $e_1, ..., e_{n+1}$, such that $e_j[0, ..., n+1] = [0, ..., j, j, ..., n+1]$.

Since we introduced requirement that vertices in a simplex are ordered, and there is actually only one way of representing any k-dimensional simplex, we can define a category of simplices as:

Definition 1.0.5. A category Δ: the of which are finite linear orders (a finite linear order of length n will be denoted as $[n]$) and the morphisms are be (not-necessarily strictly) order-preserving functions.

Using this, we can finally define a simplicial set:

Definition 1.0.6. A simplicial set is an element of $Set^{\Delta^{op}}$, e.g. a functor $X : \Delta^{op} \to Set$.

To understand this definition let us see what does such functor X does. It takes object $[n] \in \Delta$ - a simplex - and maps it to a set - a set of simplices of this shape in the simplicial set. Additionally, when it takes face map $d_i : [n+1] \to [n]$, it maps it to function between sets $X(d_i) : X([n]) \to X([n+1])$ - in this way specifying which simplices of dimension n build which faces of dimension $n+1$. In case of degeneracy map $e : [n] \to [n+1]$, we get a degeneracy $X(e) : X([n+1]) \to X([n])$.

This way, we come to the definition of simplices from quite abstract point of view:

Definition 1.0.7. Simplices are the representable functors in the category of presheaves on Δ.

The beauty and power of this categorical approach comes from the possibilities of choices of the *shape category* (here - Δ). The whole construction was done for simplices and simplicial sets - but, as we see in the next chapter, the very same procedure can be employed to define the category of opetopic sets, arising the choice of $pOpe_\iota$ as the shape category.

Chapter 2

Opetopes

2.1. Opetopes - intuition

The definition of opetope below is quite formal, so it's worth to keep in mind some intuitions about them. What we are studying are *positive-to-one computads* - devices with which we seek to encode the notion of composition. Subsequently, we have a 0-dimensional cells (constants - corresponding to geometric notion of a point), 1-dimensional cells (functions - segments in geometric view) and $n \geq 1$-dimensional cells (higher-dimensional functions - surfaces, volumes, etc). The rules of constructing them become clear while one keeps that in sight.
Opetopes are geometric in nature, and the formalism behind them is very similar to that described above in the section dealing with simplicial sets.
They can also be visualized, akin to simplicial sets.[1][2]

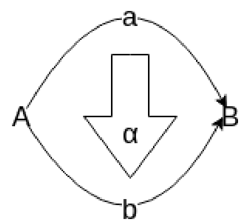


Figure 2.1: Example of a 2-dimensional opetope

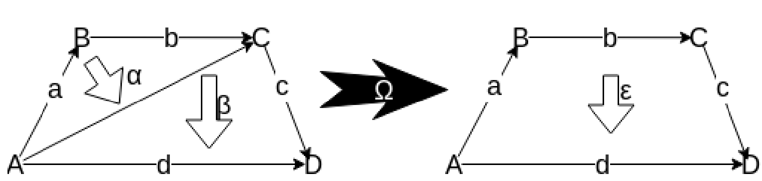


Figure 2.2: Example of a 3-dimensional opetope

[1]There is actually another method of visualizing them - and, in fact, yet another way of thinking about them - as *higher-dimensional trees*. But as this dissertation is not directly concerned with that perspective, we do not describe it here and instead recommend great visualizations in [5] or works describing theory in [6].
[2]Since there is only one possible shape of 1- and 2-dimensional opetopes, we present here examples of higher dimension.

2.2. Opetopes - definition

Definitions presented here follow [3] and [4].

Definition 2.2.1. A *positive hypergraph* is a a collection of faces $\{S_n\}_{n\in\mathbb{N}}$ (with S_0 together with only finitely many of them being non-empty), together with a collection of functions $\gamma_n : S_{n+1} \to S_n$ and a collection of total relations $\delta_n : S_{n+1} \Rightarrow S_n$, such that δ_0 is a function.

If it will be clear from context, indices of δ and γ will be omitted. We will sometimes talk about *domain* (or *preimage*) of a face α - defined as $\delta(\alpha)$ and its *codomain* (or *image*), defined as $\gamma(\alpha)$.

Definition 2.2.2. Morphisms of positive hypergraphs $\{S_n\}_{n\in\mathbb{N}} \Rightarrow \{T_n\}_{n\in\mathbb{N}}$ are families of functions $\{f_i\}_{i\in\mathbb{N}}$, such that $f_i \circ \delta = \delta \circ f_{i+1}$, $f_i \circ \gamma = \gamma \circ f_{i+1}$, and also that for any face $\alpha \in S_{k\in\mathbb{N}^+}$, function $f = f_k\big|_\alpha$ is a bijection between $\delta(\alpha)$ and $\delta(f(\alpha))$.

Definition 2.2.3. The category *pHg* is a category where objects are positive hypergraphs and morphisms are defined as above.

Using δ and γ maps, we can define two orders on faces of hypergraph:

Definition 2.2.4. A $<^+$ order is a transitive closure of $\lhd^+$ relation, defined as $a \lhd^+ b \Leftrightarrow \exists_\alpha a \in \delta(\alpha) \wedge b = \gamma(\alpha)$
A $<^-$ order is a transitive closure of $\lhd^-$ relation, defined as $a \lhd^- b \Leftrightarrow \gamma(a) \in \delta(b)$.

Definition 2.2.5. A positive hypergraph is called *positive opetopic cardinal* when it satisfies four additional conditions:

1. *Globularity*: for any $a \in S_{>1}$, functions γ and δ have to satisfy $\gamma(\gamma(a)) = \gamma(\delta(a)) - \delta(\delta(a))$ and $\delta(\gamma(a)) = \delta(\delta(a)) - \gamma(\delta(a))$.

2. *Strictness*: For every $k \in \mathbb{N}$, relation $<^+$ is a strict order. In addition, for $k = 0$, relation $<^+$ is linear order.

3. *Disjointness*: no two faces are comparable by both $<^+$ and $<^-$.

4. *Pencil-linearity*: a set of faces having common image (e.g. for any $a \in S_k$, the set $\{\alpha : a \in \delta(\alpha)\}$ is linearly ordered by $<^+$, and so is the set $\{\alpha : a = \gamma(\alpha)\}$

Definition 2.2.6. A *positive opetope* is a positive opetopic cardinal, for which $|S_n - \delta(S_{n+1})| \leq 1$ for all $n \in \mathbb{N}$.
We define a category of positive opetopes - *pOpe* - to be a full subcategory of *pHg*, with positive opetopes given as objects.

Definition 2.2.7. We say that a function h between faces of opetopes P and Q is a ι-map (or a *contraction*), if three conditions are met:

1. $dim(f(p)) \leq dim(p)$

2. It preserves codomains: for any $k+1$-dimensional face p,

$$f(\gamma^{(k)}(p)) = \gamma^{(k)}(f(p))$$

3. It preserves domains: for any $k+1$-dimensional face p
   - If $dim(f(p)) = dim(p)$, then f gives a bijection between $\delta(p) - ker(p)$ and $\delta(f(p))$ (where the kernel is defined as $\{a | dim(f(a) < dim(a)\}$ - the set of faces which are "reduced").
   - If $dim(f(p)) = dim(p) - 1$, then f gives a bijection between $\delta(p) - ker(p)$ and the singleton $\{f(p)\}$.
   - If $dim(f(p)) \leq dim(p) - 2$, then $\delta^{(k)}(p) \subset ker(p)$.

Definition 2.2.8. The category of positive opetopes $pOpe_\iota$ is a category where objects are positive opetopes and morphisms are contraction maps between them. We denote the category of presheaves $pOpe_\iota^{op} \to Set$ by $\widehat{pOpe_\iota}$. [3]

So, exactly like in the case of simplicial sets, we have a category of shapes $pOpe_\iota$ and the category of opetopic sets $\widehat{pOpe_\iota} = Set^{pOpe_\iota}$. As element of $\widehat{pOpe_\iota}$ is a contravariant functor $X : pOpe_\iota \to Set$, its value on a opetope P is a set of (abstract) opetopes of this shape in X. Also like the case of simplicial sets, we have two kinds of maps: face maps - coming from monomorphism in $pOpe_\iota$ and degeneracies - coming from epimorphisms in $pOpe_\iota$. For a monomorphism $m : Q \to P$, we get a function $X(m) : X(P) \to X(Q)$, where a value $X(m)(p)$ is a face in p. For an epimorphisms $e : Q \to P$, similarly, we get a degeneracy $X(e)(p)$ in p. An abstract opetope has finitely many faces and infinitely many degeneracies (again, like abstract simplex).

Notes: In the next chapters, we will often abuse notation to make definitions and proofs more concise. We might: identify one-element set with its element - for example, element $\delta(a)$ with a set $\{\delta(a)\}$ and $\gamma(a)$ with one-element set $\{\gamma(a)\}$; for an opetope α and face $a \in \alpha$, we will sometimes treat a as a face (e.g. element of S_k), and sometimes as opetope determined by a as its top face; call *opetopes* both elements of $pOpe_\iota$ and $\widehat{pOpe_\iota}$.
However, it will always clear from context when it happens.

[3] This is actually one of many possible ways one can define the category of opetopes. Interested reader might want to check other possible definitions in [7].

Chapter 3

Product of two opetopes

3.1. Definitions

There are a couple of useful definitions:

- We define a *dimension* of an opetope P, written as $dim(P)$ as the greatest k such that the set of faces S_k is not empty.
- For an opetope P, its *top face* is the face of maximal dimension. We will denote it by $\hat{P}$.
- The opetope P (or, as we will sometimes say, a face) is said to be *unary*, if the set $\delta(\hat{(}P))$ is a singleton.
- We say that an opetope P is a *subopetope* of Q, if there is a one-to-one map $i : P \to Q$.
- For an opetope P, we will denote the set of its k-dimensional faces by P_k, and similarly for a product of opetopes.
- We say an opetope P is globular, if it has only unary faces in all dimensions > 0.

3.2. Algorithm

Construction of the product $R = P \times Q$ is done by a double induction:

(#1) over pairs of $(dim(P), dim(Q))$ with ordering $(a, b) \leq (c, d) \Leftrightarrow (a \leq c \wedge b \leq d) \vee (a \leq d \wedge b \leq c))$

(#2) Over dimensions of the faces in R

Definition 3.2.1. Given opetopes P and Q, the construction of $P \times Q$ works as follows:

1. Base case of (#1): if $dim(P) \leq 1$ and $dim(Q) \leq 1$, add $P_0 \times Q_0 + P_0 \times Q_1 + P_1 \times Q_1 + P_1 \times Q_1$ to set of faces.
2. Induction step of (#1): without loss of generality, suppose $dim(P) \geq 2 = k_1$ and $dim(Q) = k_2$. Let us denote $\max(k_1, k_2)$ by k.

(a) Base case of (#2): we compute *initial set of faces* - a set of faces of dimension $< k$.

$$S = \{F_p \times F_q : F_p \in P_l, F_q \in Q_m, l, m < k\}$$

We know how to do that by induction (#1).

(b) Induction step of (#2): we have to construct a face F of dimension $m \geq k$ by specifying its $\delta(F), \gamma(F)$ and faces $p(F) \in P, q(F) \in Q$. Since, by induction, we have already constructed every face of dimension $< m$ in R, we try to construct F from its potential domains and codomains. We know that $p(F) = P$ and $q(F) = Q$, so now we have to check if any particular combination of previously constructed $m-1$ dimensional faces as taken as $\delta(F)$ and any particular $m-1$-dimensional face taken as $\gamma(F)$ form a valid face - by explicitly checking that they satisfy axioms and that images of such a new face under p and q are indeed $\hat{P}$ and $\hat{Q}$ (since in some situations, this can be not true).
This can actually be done in more efficient manner: utilizing DFS search, we try to build the face "from the end to the beginning" - e.g., first adding to a set T last (with respect to $<^+$ order) face $\gamma(\gamma(F))$ and then by DFS:

i. adding face a such that $\gamma(a) \in T$
ii. setting $T := T \cup \delta(a) - \gamma(a)$

until a set of potential a's is empty or a correct product face is found. Since this happens only if T is equal to $\delta(\gamma(F))$, from globularity, we can limit checking if current set of faces in $\delta(F)$ and together with the face considered to be $\gamma(F)$ form a valid face in the product only to this situation.
Additionally, we can limit the number of possibilities by utilizing $<^+$ order - we don't add a face a if any of the $\delta(a)$ is less than $\delta(\gamma(F))$ (again, with respect to $<^+$ order) - because if we added such face, then it would be never possible to reduce T to $\delta(\gamma(F))$, because with each added face a, we don't increase any minimal element of T.
Details on how to implement this, compute the $<^+$ order, etc., are provided in the Python code in the attachment **A**.

3.3. Properties

There are a few properties of this construction that are left to prove, to see that it is correct.

- Does compute every face in the product?
  What we are doing in the algorithm, is iterating over previously found faces of dimension n to produce a face of dimension $n+1$. In this process, we are interested in finding only non-degenerated faces (since all other faces will be just degeneracies of these). But, this whole process would not be correct, if it was the case that some non-degenerated faces in the product can themselves have degenerated faces. In the section 3.3.1 we prove that it can't happen, so the algorithm is indeed correct.

- Does it end on valid input?
  If we prove that there is only finitely many non-degenerated faces in the product, and every face is returned (by the preceding section), it will mean that the algorithm ends in finite time. This will be done in subsection 3.3.2.

3.3.1. Non-degenerated faces lemma

Definition 3.3.1. For an opetopic set X (element of the category $\widehat{pOpe_\iota}$), we say that an abstract opetope $x \in X(P)$ is a *degenerated face*, if there is some $y \in X(Q)$ and a map $e : P \to Q$ such that $X(P)(y) = x$ and e is epimorphism that is not a monomorphism. In such situation, we call x a degeneration of y. A face is *non-degenerated*, if it is not a degeneration of any other face.

In particular, if X is a representable functor represented by $P \in pOpe_\iota$, non-degenerated faces of X are monomorphisms $m : Q \to P$ in $pOpe_\iota$.

So, the difficult part here is that even if x and y are degenerated faces of $Hom(-, P)$ and $Hom(-, Q)$, they can be not degenerated as a pair (x,y) in $Hom(-, P) \times Hom(-, Q)$.

In a product of two opetopic sets $P \times Q$ (actually, a product $Hom(-, P) \times Hom(-, Q)$), opetopes of shape $R \in pOpe_\iota$ are pairs of maps $(p : R \to P, q : R \to Q)$. If $m : R' \to R$ is a monomorphism (in $pOpe_\iota$), then the opetope (p, q) has a face R' in $(p \circ m, q \circ m)$.
What we are showing in this section is that if (p, q) is not degenerated in $P \times Q$, then $(p \circ m, q \circ m)$ is not degenerated too.

Definition 3.3.2. An *ideal* I is a set of faces of dimension ≥ 1 in opetope P such that:

1. $\gamma(a) \in I$ if and only if $\delta(a) \subset I$
2. If $\gamma(a) \in I$, then $a \in I$
3. If $a \in I$, then $|\delta(a) - I| \leq 1$

Definition 3.3.3. For an ideal I, we say a face $u \in I$ is a *divisor* in I, if it is a unary face that is not in the domain of any other unary face and is not in the codomain of any other face, and has maximal dimension amongs these kind of faces.

Definition 3.3.4. We can now define an ideal I of P *generated by* a unary face $u \in P$, which we denote by $I(u)$. We set

$$I^0(u) = \{u\}$$

and inductively

$$\begin{aligned} I^{n+1}(u) = \{\alpha | \alpha \in I^n(u) \vee \\ \exists_{v \in I^n} \gamma(\alpha) = v \vee \\ \exists_{v \in I^n} \gamma(v) \in I^n \wedge \alpha \in \delta(v)\} \end{aligned}$$

defining $I(u)$ as

$$I(u) = \bigcup_{n \in \mathbb{N}} I^n(u)$$

In other words, it is closure of the set $\{u\}$ under ideal axioms. The $I(u)$ is indeed an ideal: this definition is trivially correct with respect to the first and second axiom (since we're taking explicit closure here), so only the status of the third axiom is interesting here. Suppose there were a face $\alpha \in I^{n+1}(u)$, such that $|\delta(\alpha) - I(u)| \geq 2$, so there are two different faces $a_1, a_2 \in \delta(\alpha)$ and $a_1, a_2 \notin I^n$. But α has been added to $I(u)$ because of three possible reasons:

- it was equal to u. But the procedure of generating $I(u)$ in each step only adds faces of higher dimension to I, and u was unary - contradiction.

- $\gamma(\alpha)$ was in some I^n. But this means that in the same step, all faces from $\delta(\alpha)$ were added - contradiction.
- there was some face v such that $c := \gamma(v)$ was in I^{n+1} and $\alpha \in \delta(v)$. But if c was in I^{n+1}, then $|\delta(c) - I^n| \leq 1$. On the other hand, the set $\{w|w <^+ a_1\} \cap \delta(c)$ has at least one element, so does the set $\{w|w <^+ a_2\} \cap \delta(c)$, and their intersection is empty (otherwise, it would violate pencil linearity). But this leads to contradiction with assumption that $|\delta(c) - I^n| \leq 1$. .

Of course, if the face u is a divisor in P, then $I(u) = \{u\}$.

Lemma 3.3.1. There exists a divisor face in every ideal I.

Proof. The ideal I is non-empty, so there is a minimal (in terms of dimension) face $\alpha \in I$. Then α is unary - because if it is not, then we know that $\gamma(\alpha) \notin I$ and that there are at least two different faces $a_1, a_2 \in \delta(\alpha)$. None of them is in I, because we assumed α is minimal w.r.t. dimension, so from the third axiom of the ideal, $|a_1, a_2| \leq 1$ we get a contradiction.
The set $t = \{a : a \in I \wedge a \text{ is unary}\}$ is non-empty; let us denote by T the set of elements of t of maximal dimension. We take the element α to be a minimal in terms of $<^+$ element of T. It is not codomain of any other face: let us assume that there is indeed some β, such that $\alpha = \gamma(\beta)$. Then there have to be some unary face $\alpha' \in \delta(\beta)$ - this is because of the globularity axiom: but $\alpha' <^+ \alpha$ and we get a contradiction. □

Definition 3.3.5. A kernel of a map $f : P \to Q$ is a set of faces in P that have been "reduced". It is defined by $ker(f) = \{a \in P : dim(f(a)) < dim(a)\}$.

Definition 3.3.6. For a divisor face $u \in P$ we define *an opetope P divided by u*, written as P/u (the set of faces $\{S'_n\}_{n\in\mathbb{N}}$, set of maps $\gamma'_{n\in\mathbb{N}}$ and $\delta'_{n\in\mathbb{N}}$), together with a map $i : P \to P/u$.

Let us denote element of the singleton $\delta(u)$ by u_d and face $\gamma(u)$ by u_c.
Faces of P/u are all faces of P except u [1], additionally divided by equivalence relation identifying u_c and u_d. If u was in the domain of any other face $\alpha \in P$, then $\delta'(\alpha) = \delta(\alpha) - \{u\}$ - it is possible, because $|\delta(\alpha)| \geq 2$, since u is a divisor.
We set $\gamma' = \gamma$, since u was not in the codomain of any face. The opetopes axioms are satisfied:

1. Globularity: for any $a \in P, a \neq u$ the maps δ' and γ' have to satisfy

$$\gamma'(\gamma'(a)) = \gamma'(\delta'(a)) - \delta'(\delta'(a))$$

   Since u was not in the image of any face, γ maps are not changed after division, so we have to prove

$$\gamma(\gamma(a)) = \gamma(\delta'(a)) - \delta'(\delta'(a))$$

   If $u \notin \delta(a)$, then $\delta'(a) = \delta(a)$ we get

$$\gamma(\gamma(a)) = \gamma(\delta(a)) - \delta'(\delta(a))$$

   and no matter if $u \in \delta'(\delta(a))$, the equality holds.

   If $u \in \delta(a)$ then $\delta'(a) = \delta(a) - \{u\}$ and

$$\gamma(\gamma(a)) = \gamma(\delta(a) - \{u\}) - \delta'(\delta(a) - \{u\})$$

[1] $S'_n = S_n$ if $u \notin S_n$ and $S'_n = S_n - \{u\}$ if $u \in S_n$

Additionally, $\gamma(\delta(a)-\{u\}) = \gamma(\delta(a))-\{u_d\}$, and since u was a singleton, $\delta'(\delta(a)-\{u\}) = \delta(\delta(a)) - u_d$. This gives us final result

$$\gamma(\gamma(a)) = \gamma(\delta(a)) - \{u_c\} - (\delta(\delta(a) - \{u_d\}))$$

because $u_d = u_c$.

Similarly, one proves that the second condition

$$\delta(\gamma(a)) = \delta(\delta(a)) - \gamma(\delta(a))$$

also holds.

2. Strictness: Relation $<^+$ is still a strict order: irreflexivity, assymetricity and transitivity obviously hold after removing u, and since u_d and u_c were consecutive elements of $<^+$, it does not break anything too.

3. Disjointness: removal of elements of two disjoint orders obviously can't make them overlap.

4. Pencil linearity: with respect to u: it doesn't change because it is just a removal of an element from an order.
With respect to u_c and u_d: let us define $g_c = \{\alpha : u_c = \gamma(\alpha)\}$ and similarly $g_d = \{\alpha : u_d = \gamma(\alpha)\}$. Then $g_c <^+ u <^+ g_d$, so after identifying u_c and u_d faces having them in the domain are comparable by $<^+$, so this doesn't break pencil linearity in this case. In the same spirit, we can set $d_c = \{\alpha : u_c = \delta(\alpha)\}$ and $g_d = \{\alpha : u_d = \delta(\alpha)\}$. Then $d_c <^+ u <^+ d_d$, so identifying u_c and u_d doesn't break anything here too.

Definition 3.3.7. For any ideal I in opetope P, we define an object representing P *divided by* I, written as P/I, together with a morphism $i : P \to P/I$.
Starting from $n = 0$, set $I_0 = I$ and $P_0 = P$. Then, until I_n is not empty, we proceed by defining u_n to be some divisor face in I (it exists from the lemma). Thus, we can divide P_n by u_n getting $i_n : P_n \to P_n/u_n = P_{n+1}$. We proceed by setting $I_{n+1} = i_n(I_n)$ and $P_{n+1} = P_n/u_n$. I_{n+1} is an ideal in P_n/u_n. Setting $J = I_{n+1}, u = u_n, \{u_d\} = \delta(u), u_c = \gamma(u)$:

- the condition $\gamma(\alpha) \in J \Rightarrow \alpha \in J$ holds trivially after removing u (it not equal to any codomain), so the only problem would be if u_d was in J, but u_c not (because that could lead to problems with some face that u_c was in the domain of) or if u_c was in J, but u_d not. However, none of these situations can happen, because of first axiom of ideals.

- the condition $\delta(\alpha) \in J \iff \gamma(\alpha) \in J$ still holds after removing u, because it was a divisor face, and identifying u_c with u_d also doesn't change anything, because their status of being (or not being) in J was the same.

- the condition $\alpha \in J \Rightarrow |\delta(\alpha)-J| \leq 1$ holds after removing and identifying faces, because then the inequality can only be stronger.

This process of divisions will stop eventually, ending on some N, because after taking image by i_n, no face of P has higher dimension, so there can be no more faces of maximal dimension then in I_n, and we eliminated one such face - namely, u_n.
Then, we set P/I to be P_N, and $i : P \to P/I$ to be $i_N \circ ... \circ i_1$.

Corollary 3.3.2. For every ideal I there is a (ι-)map $f : P \to P/I$, such that the kernel of f is I.

Proof. Directly follows from the definition of P/I. □

We say that the map f factorizes through a sequence $(u_1, ..., u_n)$.

Lemma 3.3.3. For every (ι-)map $f : P \to Q$, kernel $ker(f)$ is an ideal.

Proof. Direct check of the axioms of ideal. For any face $a \in P$:

1. If $\gamma(a) \in ker(f)$, then $dim(f(\gamma(a))) < dim(\gamma(a))$. This gives us

   $$dim(f(a)) - 1 = dim(\gamma(f(a))) \leq dim(f(\gamma(a))) < dim(\gamma(a)) \leq dim(a) - 1 = dim(a) - 1$$

   Adding 1 to both sides of the inequality yields $dim(f(a)) < dim(a)$, and that means $a \in ker(f)$.

2. Similarly, we prove that if $\gamma(a) \in ker(f)$, then for any $d \in \delta(a)$ also $d \in ker(f)$.

3. From the definition of ι-map, if $dim(f(a)) = dim(a) - 1$, the set T is a singleton, and if $dim(f(a)) < dim(a) - 1$, this set is empty.

□

Corollary 3.3.4. A set of faces in P is an ideal if and only if it is a kernel of some ι-map.

Lemma 3.3.5. If the face $u \in P$ is a divisor, and the kernel of $f : P \to Q$ contains u, then there is a h, such that the following diagram commutes:

$$\begin{tikzcd} P \arrow[r, "f"] \arrow[d, "i"] & Q \\ P/u \arrow[ur, "h"] & \end{tikzcd}$$

We say that f factorizes through P/u.

Proof. Using the lemma about correspondence between ideals and kernels, we can say that $ker(f) = I$ and I is an ideal, $u \in I$. We can then first divide by u, getting P/u and a map $h_0 : P/u \to P/I = Q$, and then by the $h_0(I)$. □

Lemma 3.3.6. If T is a face in the product $P \times Q$, then if there is a degenerated face $i : S \to T$, then T is also degenerated.

Proof. If S is degenerated, then $i : S \to T$ has a non-empty kernel - ideal I. Then $f : S \to S/I$ factorizes through $(u_1, ..., u_n)$. Let us take $u = u_1$. By the 3.3.5 lemma, $\langle p, q \rangle : S \to P \times Q$ factorizes through S/u, giving us commuting diagram

$$\begin{tikzcd} S \arrow[r, "\langle p,q \rangle"] \arrow[d, "i_s"] & P \times Q \\ S' = S/u \arrow[ur, "h_s"] & \end{tikzcd}$$

and, by the same argument, we get

$$\begin{tikzcd} S \arrow[r, "i"] \arrow[d, "i_s"] & T \\ S' \arrow[ur, "h"] & \end{tikzcd}$$

Let us take ideal J generated by u (technically - $i(u)$) in T. Because J is generated by u, J is a subset of a $ker(\langle p,q\rangle)$, so $\langle p,q\rangle : T \to P \times Q$ factorizes through a T/J, producing diagram

$$\begin{tikzcd}
T \arrow[r, "\langle p,q\rangle"] \arrow[d, "i_t"] & P \times Q \\
T/J \arrow[ur, "h_t"] &
\end{tikzcd}$$

There is also a map $g : S/u \to T/J$ from lemma 3.3.5 applied to the map h. Putting this all together gives us final diagram

$$\begin{tikzcd}
S \arrow[r, "i"] \arrow[d, "i_s"'] & T \arrow[d, "i_t"'] \arrow[ddr, bend left, "\langle p,q\rangle"] & \\
S' \arrow[ur, "h"] \arrow[r, "g"] \arrow[drr, bend right, "h_s"'] & T/J \arrow[dr, "h_t"] & \\
 & & P \times Q
\end{tikzcd}$$

So, T is indeed degenerated face in the product. $\square$

This gives us the result:

Lemma 3.3.7. Every face in the product will be returned by the construction.

Proof. Suppose a face r of dimension l is not returned by the algorithm, and that all faces of dimension lower than l are returned. Then $l \geq 2$, because faces of dimension $0, 1$ are explicitly enumerated in the algorithm. But, in the induction step (#2), in step of dimension l, algorithm considers all possible non-degenerated faces constructed from faces of dimensions $l-1$, and since non-degenerated faces can only have non-degenerated (sub)-faces themselves - this follows from the previous lemma - the algorithm considers r too. $\square$

3.3.2. Finiteness of the product

Lemma 3.3.8. For any P and Q, there are only finitely many non-degenerated faces in $P \times Q$.

Proof. Proof by induction over pairs $(dim(P), dim(Q))$ ordered by transitive closure of order $(a,b) < (c,d) \Leftrightarrow (a < c) \wedge (b < d)$.
For pairs $(\star, k)$ (and analogously $(k, \star)$) for $k \leq 1$ this fact is obvious (but formal proof can be found in [4]).
For a $n \in \mathbb{N}$, let us denote family of faces of dimension n in product by R_n. In first step, we will prove that for every n, $|R_n| < \aleph_0$.
Suppose that for some n, there is an infinite number of non-degenerated faces of dimension n - suppose that n_0 is a lowest such dimension, and let us denote countable subset of them by $\{r_i\}_{i\in\omega} \subset R_{n_0}$. The case $n_0 < max(dim(P), dim(Q))$ cannot happen.
There are only finitely many combinations of faces in R_{n_0-1}, so some two (different) faces $r, r' \in R_{n_0}$ have to satisfy equalities $\delta(r) = \delta(r')$ and $\gamma(r) = \gamma(r')$. At the same time, $p(r) = p(r') = \hat{P}$ and $q(r) = q(r') = \hat{Q}$. But these facts together would mean that r and r' are the same face.
Second step of the proof is showing that there is some number $n_1 \in \mathbb{N}$, such that for every $n > n_1$, $|R_n| = 0$. Let us choose $n > max(dim(P), dim(Q))$ fixed.

First observation is that there are no unary faces above the dimension $max(dim(P), dim(Q)$ - because then all faces from $\{a, \gamma(a), \delta(a)$ would map to the same (top) face in both P and Q, so it would make a a degenerated face.
Second observation is that n chosen above satisfies inequalities

$$\max_{\alpha \in S_n}(\delta(\alpha)) \leq \delta(\gamma(\alpha)) - 1 \leq \max_{a \in \delta(\alpha) \cap \{\gamma(\alpha)\}}(\delta(a)) - 1$$

Because α is arbitrary, we have the following inequality

$$\max_{\alpha \in R_n}(\delta(\alpha)) \leq \max_{\alpha \in R_{n-1}}(\delta(\alpha)) - 1$$

It means that the function $k \longmapsto \max_{\alpha \in S_k}(\delta(\alpha))$ is strictly decreasing on sets of faces S_k, and since it takes values in $\mathbb{N}$, it has to be equal to 0 for some n_1 and all next sets of faces $S_N, N > n_1$ have to be empty. $\square$

Chapter 4

Implementation

4.1. Code description

As attachments to this dissertation, we present two implementations of the algorithm.

4.1.1. Python

At first, Python implementation was created. There is a noticeable overhead of using interpreted language, but it pays off in terms of ease of results inspection, as well as code optimization. In particular, this implementation uses $<^+$ order for search-pruning, and relies on immutable structures in order to use memoization, in effect never requiring computing same function call twice.
The solution is made of two files - `Opetope.py` implements representation of opetopes (Opetope class) and product faces (Face class). `Product.py` file contains code responsible for actual product computation and some auxiliary data structures and functions dealing with $<^+$ order.
Besides computing product representation, the code also checks if the result is contracible. It uses horn filling algorithm to do that.
Code is documented and should be easily understandable after reading this dissertation.

During the development, we used a following notation for describing opetopes[1].
An opetope with P with $\delta(P) = \{d_1, ..., d_n\}$ and $\gamma(P) = c$ is described as $P : [d_1, ..., d_n] \to c$. In many cases we skip P and write just $[d_1, ..., d_n] \to c$, but it should be clear from context which opetope are we referring to. If the domain of P is a singleton, we sometimes drop "["" and "]"" and write just $P : d \to c$.
A 0-dimensional face R in the product of two opetopes $P \times Q$ is described as a pair $(p(R), q(R))$, where $p(R)$ is the point in P and $q(R)$ is the point in Q that R maps into. If the dimension of R is greater than 0, its representation is analogous to the representation of opetope, as described above (with 0-dimensional faces described as pairs).
To make notation more concise, we assume that two-small-letter names with a succeeding number (for example: xy_1) refer to a one-dimensional opetope with input and output points described by the letters (so, xy is an abbreviation for $xy_1 : [x] \to y$). If there are just two small letter we assume default number 1 (so, xy should be read as xy_1).
Newest version is always available on `https://github.com/inexxt/opetopes`

[1] as the default string representation of objects in `Python`

4.1.2. Idris

Later, algorithm was reimplemented from scratch in Idris, using dependent types. They allow to explicitly express various constraints `Opetope` type has to satisfy - the implementation mainly deals with dimensions correctness. As this implementation was not the main focus of this work, it lacks any optimizations, thus being significantly slower than the Python version. Main definitions of opetope type and product face type are respectively in `Opetope.idr` and `Face.idr`. Helper functions are defined in `OpetopeUtils.idr` and `FaceUtils.idr`, and main algorithm - in `Product.idr`.
Newest version is always available on `https://github.com/inexxt/opetopes-idris`.

4.2. Results description

We run small, medium, and large experiments. One of the main test sources were products in the form of $(\star \xrightarrow{\star} \star) \times P$, since they can be efficiently constructed using an algorithm developed in [4].
Below, we report a table describing sizes of various products.

Table 4.1: Number of faces in products of globular opetopes

	0-dimensional	1-dimensional	2-dimensional
0-dimensional	1		
1-dimensional	3	11	
2-dimensional	5	25	101
3-dimensional	7	51	451
4-dimensional	9	101	2197
5-dimensional	11	199	11175

Considering 2-dimensional opetopes, the simplest case is that 1-dimensional cells in the domain are linearly ordered. In the table below, we report the number of faces for products of such opetopes.

Table 4.2: Number of faces in products of simple 2-dimensional "linear" opetopes

	$\lvert\delta(\star)\rvert = 5$	$\lvert\delta(\star)\rvert = 7$	$\lvert\delta(\star)\rvert = 9$
$\lvert\delta(\star)\rvert = 5$	101		
$\lvert\delta(\star)\rvert = 7$	157	271	
$\lvert\delta(\star)\rvert = 9$	213	409	981
$\lvert\delta(\star)\rvert = 11$	269	571	2233
$\lvert\delta(\star)\rvert = 13$	325	757	4469
$\lvert\delta(\star)\rvert = 15$	381	967	7993
$\lvert\delta(\star)\rvert = 17$	437	1201	13109
$\lvert\delta(\star)\rvert = 19$	493	1459	20121
$\lvert\delta(\star)\rvert = 21$	549	1741	29333

We define names for the following opetopes:

- Glob: $[\alpha : ab_1 \to ab_2] \to [\beta : ab_1 \to ab_2]$

- Glob-Glob: $[\alpha : ab_1 \to ab_2, \beta : ab_2 \to ab_3] \to [\epsilon : ab_1 \to ab_3]$
- Glob-Glob-Glob: $[\alpha : ab_1 \to ab_2, \beta : ab_2 \to ab_3, \epsilon : ab_3 \to ab_4] \to [\zeta : ab_1 \to ab_4]$
- Triangle: $[\alpha : [ab, bc] \to ac] \to [\beta : [ab, bc] \to ac]$
- Triangle-Glob: $[\alpha : [ab, bc] \to ac_1, \beta : ac_1 \to ac_2] \to [\epsilon : [ab, bc] \to ac_2]$
- Glob-Triangle: $[\alpha : ab_1 \to ab_2, \beta : [ab_2, bc] \to ac] \to [\epsilon : [ab_1, bc] \to ac]$
- Square: $[\alpha : [ab, bc, cd] \to ad] \to [\beta : [ab, bc, cd] \to ad]$

Table 4.3: Number of faces in products of 3-dimensional opetopes

	Glob	Triangle	Glob-Glob
Glob	5795		
Triangle	9471	19097	
Glob-Glob	10359	17287	20085
Square	13147	32051	24215
Triangle-Glob	14035	28609	27013
Glob-Triangle	14979	28801	29925
Glob-Glob-Glob	15771	26799	33387

Chapter 5

Summary

In this work, we presented an algorithm for construction $P \times Q$. It is, however, not a really efficient one - and this is strongly magnified here. Products of small opetopes can be really big - for example, a product of 2-dimensional, "linear", 11-face opetope with the same kind, 13-face one has over 70000 faces (which actually makes it the biggest product computed by us to date).
However, it was not really designed to serve this purpose - all it had to do is to allow for checking hypothesis quicker, allowing one to construct more advanced algorithm basing on verification provided by this one.
Implementation-wise, one low-hanging optimization route was not explored - namely, caching the result of product computation on the level of *shapes* being computed. For example, a product of two arrows (i.e. $(\star \xrightarrow{\star} \star) \times (\star \xrightarrow{\star} \star)$) is computed very often, resulting in significant slowdown. It would be possible to construct *templates* of shapes, so that the program, instead of calling computing product function all over again, would simply fill-in appropriate template. We did not implemented that because of lack of time, but it should not prove to be particularly challenging task.

Appendix A

Code in Python

A.1. Code

Listing A.1: Opetope.py

```python
from typing import Set, Iterable


def flatten(ss):
    """
    Flatten a list - [[a, b], [c, d]] becomes [a, b, c, d]
    :param ss: list
    """
    return [x for s in ss for x in s]


# Generating unique ids not using RNG
ids = []


def generate_id(op: 'Opetope'):
    if not op.level:
        return ""

    ids.append(len(ids))
    return str(ids[-1])


def masks(n):
    """
    Generate all possible bit masks of length n
    :param n: length
    """
    if n == 1:
        return [[True], [False]]
    else:
        ms = masks(n - 1)
        return flatten([[[True] + m, [False] + m] for m in ms])


def unescape(x):
    """
    Remove ' and " from the string
    :param x: string
    """
    return x.replace("'", "").replace('"', "")


def first(iterable, default=None):
    """Return the first element of an iterable or the next element of a
    generator; or default.
    From norvig.com"""
    try:
```

```
            return iterable[0]
    except IndexError:
        return default
    except TypeError:
        return next(iterable, default)


class NegCounter():
    """
    I had to implement my own Counter class, because the default one doesn't
    support negative values... Or else, it does, but not consistently.
    """

    def __init__(self, obj=None):
        self.counts = {}
        if obj:
            if isinstance(obj, dict):
                self.counts = {k: v for k, v in obj.items()}
            elif isinstance(obj, NegCounter):
                self.counts = {k: v for k, v in obj.counts.items()}
            elif isinstance(obj, Iterable):
                for t in obj:
                    self.counts[t] = self.counts.get(t, 0) + 1

    def __add__(self, other):
        return NegCounter({
            x: self.counts.get(x, 0) + other.counts.get(x, 0)
            for x in set(self.counts.keys()) | set(other.counts.keys())
        })

    def __sub__(self, other):
        return NegCounter({
            x: self.counts.get(x, 0) - other.counts.get(x, 0)
            for x in set(self.counts.keys()) | set(other.counts.keys())
        })

    def __or__(self, other):
        return NegCounter({
            x: self.counts.get(x, 0) + other.counts.get(x, 0)
            for x in set(self.counts.keys()) | set(other.counts.keys())
        })

    def __and__(self, other):
        return NegCounter({
            x: self.counts.get(x, 0) + other.counts.get(x, 0)
            for x in set(self.counts.keys()) & set(other.counts.keys())
        })

    def is_empty(self):
        return all(not v for v in self.counts.values())

    def __iadd__(self, other):
        return NegCounter({
            x: self.counts.get(x, 0) + other.counts.get(x, 0)
            for x in set(self.counts.keys()) & set(other.counts.keys())
        })

    def __isub__(self, other):
        return NegCounter({
            x: self.counts.get(x, 0) - other.counts.get(x, 0)
            for x in set(self.counts.keys()) & set(other.counts.keys())
        })

    def __getitem__(self, item):
        if item not in self.counts:
            self.counts[item] = 0
        return self.counts[item]

    def __setitem__(self, key, value):
        self.counts[key] = value

    def __repr__(self):
        return self.counts.__repr__()

```

```python

class Opetope:

    __slots__ = [
        "name", "level", "ins", "out", "_shape", "_str", "_all_subopetopes",
        "_all_subouts", "id", "splus_order"
    ]

    def __init__(self, ins=(), out=None, name=""):
        """
        :param ins: An iterable of opetopes one level lower
        :param out: A single opetope one level lower
        :param name: Name of this opetope - if not provided, a unique new one
        will be created
        """
        self.name = name

        if out:
            assert isinstance(out, Opetope)

            # check that levels are ok
            self.level = out.level + 1
            assert Opetope.match(ins, out, self.level)

            # check that lower level opetopes really "match"
            self.ins = tuple(ins)
            self.out = out

        else:
            self.level = 0
            self.ins = ()
            self.out = -1

        self.id = name if name else generate_id(self)

        # pre-calculating attributes
        self._shape = self.calculate_shape()
        self._str = self.calculate_to_string()
        self._all_subopetopes = frozenset(self.calculate_all_subopetopes())
        self._all_subouts = frozenset(self.calculate_all_subouts())

    @staticmethod
    def match(ins, out, level) -> bool:
        """
        Check if the out and ins provided match together, to create a new
        level-dimension opetope
        """

        if level == 0:
            return ins == () and out == None

        if level == 1:
            return out.level == 0 and len(ins) == 1 and ins[0].level == 0

        if not all([i.level == out.level
                    for i in ins]) or out.level + 1 != level:
            return False

        ins_of_out = NegCounter(out.ins)
        for opetope in ins:
            ins_of_out = ins_of_out - NegCounter(opetope.ins)
            ins_of_out = ins_of_out + NegCounter({opetope.out})
        ins_of_out = ins_of_out - NegCounter({out.out})
        return ins_of_out.is_empty()

    def __str__(self) -> str:
        return self._str

    def __repr__(self) -> str:
        return self._str

    def is_unary(self) -> bool:
        """
        Check if the opetope is unary, eg it has exactly one face in the domain
```

```python
        These kind of opetopes can be then degenerated
        :return:
        """
        return len(self.ins) == 1

    def calculate_shape(self):
        if not self.level:
            return "*"

        return "({} -> {})".format([i._shape for i in self.ins],
                                   self.out._shape)

    def calculate_to_string(self) -> str:
        """
        Return string representation of the opetope
        :param remove_names: This is used if one want's to have an "abstract"
        representation of an opetope - just the shape
        """
        if not self.level:
            return self.name

        return unescape("({}: {} -> {})".format(
            self.name, sorted([i._str for i in self.ins]), self.out._str))

    def calculate_all_subopetopes(self) -> 'FrozenSet[Opetope]':
        if not self.level:
            return frozenset({self})

        return frozenset(flatten([
            o.all_subopetopes() for o in self.ins
        ])) | self.out.all_subopetopes() | frozenset({self})

    def calculate_all_subouts(self) -> 'FrozenSet[Opetope]':
        if not self.level:
            return frozenset()
        return frozenset({self.out}) | frozenset(
            flatten([o.all_subouts() for o in [*self.ins, self.out]]))

    def all_subopetopes(self):
        return self._all_subopetopes

    def all_subouts(self):
        return self._all_subouts

    def shape(self, remove_names=True):
        return self._shape

    @staticmethod
    def from_shape(shape):
        # return Opetope with shape specified
        pass

    def __eq__(self, other):
        return str(self) == str(other)

    def __hash__(self):
        return hash(self._str)

    @staticmethod
    def is_valid_morphism(op1: 'Opetope', op2: 'Opetope') -> bool:
        """
        Check that op1, with vertices colored (named) by vertices of op2, is
        a valid contraction to ope. One problem is that we can't use the
        top-level name
        :param op1:
        :param op2:
        :return:
        """

        # contract all things in op1
        def contract(op):
            if not op.level:
                return op
            out = contract(op.out)
```

```
            ins = [contract(i) for i in op.ins]
            ins = [i for i in ins if i.level == out.level]

            if all([i._str == out._str for i in ins]):
                return contract(op.out)
            return Opetope(ins=ins, out=out, name=op.name)

        op1.name = op2.name
        return contract(op1)._str == op2._str

    def is_non_degenerated(self):
        # basically not having loops
        if not self.level:
            return True
        if self.level == 1:
            return self.ins[0] != self.out
        else:
            return all(i.is_non_degenerated() for i in [*self.ins, self.out])


class Face(Opetope):

    __slots__ = ["p1", "p2", "_str_full"]

    def __init__(self,
                 p1: Opetope,
                 p2: Opetope,
                 ins: 'Iterable[Face]' = (),
                 out=None,
                 name=""):
        self.p1 = p1
        self.p2 = p2
        self._str_full = ""

        super().__init__(ins=ins, out=out, name=name)

        self._str_full = self.calculate_to_string(full=True)

    def calculate_to_string(self, full=False) -> str:
        if full:
            if not self.level:
                return "{}{}".format(self.p1, self.p2)

            return "{}{}{}{}{}".format(
                self.p1, self.p2, "".join(
                    sorted([i._str_full for i in self.ins])),
                self.out._str_full, self.name)
        else:
            return "({}, {})!{}\n".format(self.p1._str, self.p2._str,
                                          self.level)

    @staticmethod
    def verify_construction(p1: Opetope,
                            p2: Opetope,
                            ins: 'Iterable[Face]' = (),
                            out=None,
                            name="") -> bool:
        if not Opetope.match(ins, out, out.level + 1):
            return False

        face = Face(p1, p2, ins, out, name)

        def get_pxs(f: 'Face', px) -> Opetope:
            if not f.level:
                return Opetope(name=px(f).name)

            out = get_pxs(f.out, px)
            ins = [get_pxs(i, px) for i in f.ins if i.level == out.level]
            return Opetope(ins=ins, out=out, name=px(f).name)  # (*)

        op1 = get_pxs(face, lambda x: x.p1)
        op2 = get_pxs(face, lambda x: x.p2)

        # FIXME remove these
```

```python
        op1.name = "name"
        op2.name = "name"

        # We have to check here if this is a valid projection
        # eg if all (recursivly) faces of self, projected on p1, together
        # get us p1, and similarly p2
        if not (Opetope.is_valid_morphism(op1, p1)
                and Opetope.is_valid_morphism(op2, p2)):
            return False

        return True

    @staticmethod
    def from_point_and_point(p1: Opetope, p2: Opetope) -> 'Face':
        assert (p1.level, p2.level) == (0, 0)
        return Face(p1, p2)

    @staticmethod
    def from_arrow_and_point(p1: Opetope, p2: Opetope) -> 'Face':
        assert (p1.level, p2.level) == (1, 0)
        return Face(
            p1,
            p2,
            ins=[Face.from_point_and_point(p1.ins[0], p2)],
            out=Face.from_point_and_point(p1.out, p2))

    @staticmethod
    def from_point_and_arrow(p1: Opetope, p2: Opetope) -> 'Face':
        assert (p1.level, p2.level) == (0, 1)
        # we can't just use from_arrow_and_point
        # because the order p1, p2 is important
        return Face(
            p1,
            p2,
            ins=[Face.from_point_and_point(p1, p2.ins[0])],
            out=Face.from_point_and_point(p1, p2.out))

    @staticmethod
    def from_arrow_and_arrow(p1: Opetope, p2: Opetope) -> 'Face':
        assert (p1.level, p2.level) == (1, 1)
        return Face(
            p1,
            p2,
            ins=[Face.from_point_and_point(p1.ins[0], p2.ins[0])],
            out=Face.from_point_and_point(p1.out, p2.out))

    def __eq__(self, other):
        return hash(self) == hash(other)

    def __hash__(self):
        return hash(self._str_full)

    def __str__(self):
        return self._str

    def __repr__(self):
        return self._str
```

Listing A.2: Products.py

```python
import itertools

import os

try:
    from fastcache import lru_cache
except:
    from functools import lru_cache

from Opetope import Opetope, Face, flatten, NegCounter, first

from typing import Set, FrozenSet, Tuple

import pickle

all_results = set()
all_missed = []
all_not_missed = []

DEBUG = True

order = set()


def is_in_order(b, target_out):
    return all(any((bi, ti) in order for ti in target_out.ins) for bi in b.ins)


def build_possible_opetopes(op, building_blocks, P, Q):
    # build all possible opetopes which have the codomain == op
    # and are constructed only from elems

    # and proceed from here by DFS
    results = DFS(
        frozenset([op.out]), frozenset(), frozenset(building_blocks), op, P, Q)
    return results


@lru_cache(maxsize=None)
def DFS(current_ins: FrozenSet[Face], used: FrozenSet[Face],
        building_blocks: FrozenSet[Face], target_out: Face, P: Opetope,
        Q: Opetope):

    if target_out.level < 1:
        return set()

    if Face.verify_construction(p1=P, p2=Q, ins=used, out=target_out):
        new_face = Face(p1=P, p2=Q, ins=used, out=target_out)
        all_results.add(new_face)
        if DEBUG:
            print("Current face count: {}".format(len(all_results)))
        #     print(new_face.ins, new_face.out)
        #     debug_faces.add(new_face)
        return {new_face}

    # ugly hack, but points do not have themselves as outs, so it is needed
    out = lambda x: x if not x.level else x.out

    # if not, we have to iterate through all possible to use opetopes and
    # check each combination recursively
    results = set()
    for b in building_blocks - used:
        for i in current_ins:
            # if DEBUG:
            #     print("Now focusing on b: {} u: {}".format(b, i))
            if i == out(b) and i.p1 in P.all_subopetopes(
            ) and i.p2 in Q.all_subopetopes():
                if not is_in_order(b, target_out):
                    continue
                new_ins = frozenset({*current_ins, *b.ins} - {i})
                new_used = frozenset([*used, b])

```

```
                # assert len(new_used) > len(used)
                # assert len(new_blocks) < len(building_blocks)
                results |= DFS(
                    current_ins=new_ins,
                    used=new_used,
                    building_blocks=building_blocks,
                    target_out=target_out,
                    P=P,
                    Q=Q)

    return results


@lru_cache(maxsize=None)
def product(P: Opetope, Q: Opetope) -> (Set[Face], Set[Face]):

    # if DEBUG:
    #     print("Now analyzing opetopes {} and {}".format(P, Q))
    subs1 = P.all_subopetopes()
    subs2 = Q.all_subopetopes()

    # the goal is to construct big_faces - the faces which map simultaneously
    # to whole op1 and whole op2
    big_faces = set()

    # we also need small_faces - these are the ones that don't map to whole
    # op1 and whole op2 simultaneously
    small_faces = set()

    points = lambda s: {p for p in s if not p.level}
    arrows = lambda s: {p for p in s if p.level == 1}
    small_faces |= {
        Face.from_point_and_point(s1, s2)
        for s1 in points(subs1) for s2 in points(subs2)
    }
    small_faces |= {
        Face.from_arrow_and_point(s1, s2)
        for s1 in arrows(subs1) for s2 in points(subs2)
    }
    small_faces |= {
        Face.from_point_and_arrow(s1, s2)
        for s1 in points(subs1) for s2 in arrows(subs2)
    }
    small_faces |= {
        Face.from_arrow_and_arrow(s1, s2)
        for s1 in arrows(subs1) for s2 in arrows(subs2)
    }

    # going from the lowest dimension first
    s1s2 = itertools.product(subs1, subs2)
    for (s1, s2) in s1s2:  # FIXME remove sorted
        if (s1, s2) != (P, Q) and (s1.level, s2.level) not in [(0, 1), (0, 0),
                                                                (1, 0)]:
            big, small = product(s1, s2)
            # big faces from subopetope are small faces in here
            small_faces |= big | small

    add_to_splus_order(order,
                       small_faces)  # ugly but necessary non-pure function

    # minimal dimension of such a face is k = max(dim(P), dim(Q))
    k = max(P.level, Q.level)

    # induction on l - dimension of such face
    l = k

    # special case when we product two arrows and there is big face from
    # the beginning
    if P.level == 1 and Q.level == 1:
        big_faces |= {Face.from_arrow_and_arrow(P, Q)}

    # we proceed until there is no new face
    while True:
        add_to_splus_order(order,
```

```python
                           big_faces)  # ugly but necessary non-pure function

        # we have constructed all big faces of dimension < l
        # we now proceed to faces dimension l

        # the possible codomains of such a face are:
        possible_codomains = set()
        # all (l-1)-dimensional big_faces
        possible_codomains |= {f for f in big_faces if f.level == l - 1}

        if l == k:
            possible_codomains |= {
                f
                for f in small_faces if f.p1 == P.out and f.p2 == Q.out
            }

        # if dim(P) <= dim(Q), then it may be a face that maps to P and gamma(Q)
        if P.level < Q.level and l == k:
            possible_codomains |= {
                f
                for f in small_faces
                if f.p1 == P and f.p2 == Q.out and f.level == l - 1
            }
        # if dim(Q) <= dim(P), then it may be a face that maps to Q and gamma(P)
        if Q.level < P.level and l == k:
            possible_codomains |= {
                f
                for f in small_faces
                if f.p1 == P.out and f.p2 == Q and f.level == l - 1
            }

        # now, for each possible codomain, we build the opetope that contains it
        new_opetopes = set()
        for f in possible_codomains:
            # I think it is enough to build just from the stuff that has the
            # right dimension eg, equal to dim(f)
            building_blocks = {
                s
                for s in small_faces | big_faces
                if s.level == f.level and f != s
            }
            new_opetopes |= build_possible_opetopes(
                op=f, building_blocks=building_blocks, P=P, Q=Q)

        # checking for l > 1 for special case when we product two arrows
        if not new_opetopes and l > 1:
            return (big_faces, small_faces)

        big_faces |= new_opetopes

        l += 1


def transitive_reflexive_closure(relation: Set, new_elems: Set):
    closed_rel = set()
    closed_rel |= relation

    while True:
        added_elems = {(x, z)
                       for (x, y) in new_elems for (w, z) in closed_rel
                       if y == w}
        added_elems |= {(x, z)
                        for (x, y) in closed_rel for (w, z) in new_elems
                        if y == w}

        if not added_elems - closed_rel:
            break
        closed_rel |= added_elems
        new_elems |= added_elems

    closed_rel |= {(x, x) for (x, _) in closed_rel}
    closed_rel |= {(x, x) for (_, x) in closed_rel}

    return closed_rel
```

```python


def calculate_splus_order(opetope: Face) -> Set[Tuple[Face, Face]]:
    """"partial order on subopetopes of equal dimension
    x <= y, x.level == y.level =: p if there is a sequence of opetopes
                                    o_1, ... o_n of levels p + 1, such that:
     - o_(k+1).out in o_k.ins
     - y in o_n.ins
     - x == o_1.out
    + transitive-reflexive closure of said relation"""
    order = set()

    for sub_ope in opetope.all_subopetopes():
        if sub_ope.level:
            order |= {(sub_ope.out, i) for i in sub_ope.ins}
        order |= {(sub_ope, sub_ope)}

    return order


def add_to_splus_order(order, faces):
    new_elems = set()
    for f in faces:
        if not (f, f) in order:
            new_elems |= calculate_splus_order(f)

    if new_elems:
        # big computational overhead
        order |= new_elems
        order |= transitive_reflexive_closure(order, new_elems)


class Product:
    def __init__(self, p1: Opetope, p2: Opetope):
        self.p1 = p1
        self.p2 = p2

        b, s = product(p1, p2)
        self.faces = b | s
        print("Evals ", len(all_missed))

    def __repr__(self):
        c = NegCounter()
        for x in self.faces:
            c[x.level] += 1
        return [(k, c[k]) for k in sorted(c.counts)].__repr__()

    def __str__(self):
        return self.__repr__()

    def is_contractible(self):
        # horn filling

        all_faces = set(flatten(f.all_subopetopes() for f in self.faces))
        points = {k for k in all_faces if not k.level}
        # start with any point
        p = first(iter(points))
        used = {p}
        all_faces.remove(p)

        # flag indicating whenever something changed in the last loop
        flag = True

        # add face when all faces in its codomain are already added
        while flag:
            flag = False
            for f in all_faces - points:
                # if all but one faces are already added
                if sum(bool(k in used)
                       for k in f.ins) + bool(f.out in used) == len(f.ins):
                    # add the remaining face and its last face
                    used.add(f)
                    used.add(f.out)
                    used.update(set(f.ins))
```

```
                    flag = True
            all_faces -= used
        return not all_faces

    def save(self, path=""):
        if not path:
            path = os.path.join(".", "pickles",
                                f"product-{len(self.p1._all_subopetopes)}"
                                f"-{len(self.p2._all_subopetopes)}"
                                f"-{time.time()}.pickle")
        with open(path, "wb") as f:
            pickle.dump(self, f)
```

A.2. Example input and output files

The syntax of input files should be easily understood - it is just a list of faces, with their domains and codomains. Two additional parameters are:

1. `square` - if we want to compute a product of opetope with itself
2. `unique_names` - set in the case of `square True`, indicates if the names printed as output should be made unique - to avoid confusion between faces of first and second elements of the product.

Listing A.3: 11_11.yaml

```
first:
    a: []
    b: []
    c: []
    d: []
    e: []
    ab: [[a], b]
    bc: [[b], c]
    cd: [[c], d]
    de: [[d], e]
    ae: [[a], e]
    alpha: [[ab, bc, cd, de], ae]

options:
    square: True
    unique_names: True
```

Listing A.4: 2eye_2eye.yaml

```
first:
    a: []
    b: []
    ab1: [[a], b]
    ab2: [[a], b]
    alpha1: [[ab1], ab2]
    alpha2: [[ab1], ab2]
    whole: [[alpha1], alpha2]

options:
    unique_names: True
    square: True
```

Listing A.5: arrow_9g.yaml

```
first:
  a1: []
  a2: []
  a3:
  - [a2]
  - a1
  a4:
  - [a2]
  - a1
  a5:
  - [a4]
  - a3
  a6:
  - [a4]
  - a3
  a7:
  - [a6]
  - a5
  a8:
  - [a6]
  - a5
  a9:
  - [a8]
  - a7
options: {unique_names: true}
second:
  a: []
  ab:
  - [a]
  - b
  b: []
```

Appendix B

Code in Idris

Listing B.1: Opetope.idr

```
module Opetope

import Data.SortedBag as MS
import Utils as U


public export
data Opetope : Nat -> Type where
    Point : String -> Opetope Z
    Arrow : String -> Opetope Z -> Opetope Z -> Opetope (S Z)
    Face : String -> List (Opetope (S n)) -> Opetope (S n) -> Opetope (S (S n))


export
name : Opetope n -> String
name (Point s) = s
name (Arrow s _ _) = s
name (Face s _ _) = s

public export
dim : {n: Nat} -> Opetope n -> Nat
dim {n} _ = n

lemma_zero : (dim (Point "a")) = Z
lemma_zero = Refl


export
Show (Opetope n) where
    show (Point s)     = s
    show (Arrow s d c) = "(" ++ s ++ ": " ++ show [d] ++ " -> " ++ show c ++ ")"
    show (Face s d c)  = "(" ++ s ++ ": " ++ show d ++ " -> " ++ show c ++ ")"


mutual
    public export
    Eq (Opetope n) where
        (Point s1) == (Point s2)             = s1 == s2
        (Arrow s1 d1 c1) == (Arrow s2 d2 c2) = (s1, d1, c1) == (s2, d2, c2)
        (Face s1 d1 c1) == (Face s2 d2 c2)   = (s1, sort d1, c1) == (s2, sort d2, c2)

    public export
    Eq (Opetope n) => Ord (Opetope n) where
        compare (Point s1) (Point s2)             = compare s1 s2
        compare (Arrow s1 d1 c1) (Arrow s2 d2 c2) = compare (s1, d1, c1) (s2, d2, c2)
        compare (Face s1 d1 c1) (Face s2 d2 c2)   = compare (s1, sort d1, c1) (s2, sort d2, c2)

export
build_op : (n: Nat) -> Opetope n
build_op Z = Point "a"
build_op (S Z) = Arrow "b" (build_op Z) (build_op Z)
build_op (S (S n)) = Face "c" [(build_op (S n))] (build_op (S n))

export
dom : (Opetope (S n)) -> List (Opetope n)
dom (Arrow _ d _) = [d]
dom (Face _ d _) = d

export
cod : (Opetope (S n)) -> Opetope n
cod (Arrow _ _ c) = c
cod (Face _ _ c) = c

public export
OSet : Nat -> Type
```

```
OSet n = MS.SortedBag (Opetope n)

public export
Eq (OSet n) where
    a == b = (MS.toList a) == (MS.toList b)

public export
Show (OSet n) where
    show a = show (MS.toList a)

export
match : {n: Nat} -> Opetope n -> Bool
match {n=Z} _ = True
match {n=(S Z)} _ = True
match {n=(S (S k))} (Face _ ins out) =
        (all_dom `MS.union` out_cod) == (all_cod `MS.union` out_dom)
        && U.and_ (map match ins)
        && match out
    where
        all_dom : OSet k
        all_dom = MS.fromList (concat $ map dom ins)
        out_dom : OSet k
        out_dom = MS.fromList (dom out)
        all_cod : OSet k
        all_cod = MS.fromList (map cod ins)
        out_cod : OSet k
        out_cod = MS.singleton (cod out)

export
is_unary : Opetope (S dim) -> Bool
is_unary op = (length (dom op)) == 1


export
eq : {n1: Nat} -> {n2: Nat} -> Opetope n1 -> Opetope n2 -> Bool
eq {n1} {n2} op1 op2 = case decEq n1 n2 of
    Yes prf => (replace prf op1) == op2
    No _ => False

export
comp : {n1: Nat} -> {n2: Nat} -> Opetope n1 -> Opetope n2 -> Ordering
comp {n1} {n2} op1 op2 = case decEq n1 n2 of
    Yes prf => compare (replace prf op1) op2
    No _ => compare n1 n2
```

Listing B.2: OpetopesUtils.idr

```
module OpetopesUtils

import Data.SortedBag as MS
import Data.HVect as HV

import Opetope
import Utils as U

%access public export


OMap : Type
OMap = (n: Nat) -> OSet n

empty: OMap
empty n = MS.empty

singleton : {n: Nat} -> (Opetope n) -> OMap
singleton {n} x = \k => case decEq n k of
    Yes prf => MS.singleton (replace prf x)
    No _ => MS.empty

get : (n:Nat) -> OMap -> OSet n
get n om = om n

union : OMap -> OMap -> OMap
union om1 om2 = \n => MS.union (get n om1) (get n om2)

unions : (List OMap) -> OMap
unions [] = empty
unions (x::xs) = x `union` (unions xs)


subopetopes : Opetope n -> OMap
subopetopes op = case op of
    (Point x) => singleton op
    (Arrow s d c) => unions $ [singleton op, subopetopes d, subopetopes c]
    (Face s d c) => unions $ [singleton op, (unions (map subopetopes d)), subopetopes c]

subouts : Opetope n -> OMap
subouts op = case op of
    (Point x) => empty
    (Arrow s d c) => singleton c
    (Face s d c) => unions [singleton c, (unions (map subouts d)), subouts c]


is_non_degenerated : Opetope n -> Bool
is_non_degenerated op = case op of
    (Point _) => True
    (Arrow _ d c) => c /= d
    (Face _ d c) => (U.and_ (map is_non_degenerated d)) && (is_non_degenerated c)


Show OMap where
    show t = show' t 0
        where
            show' : OMap -> Nat -> String
            show' t n = if (p == MS.empty) then ""
                        else ((show p) ++ ", " ++ (show' t (n + 1)))
                where
                    p : OSet n
                    p = t n
```

Listing B.3: Face.idr

```
module Face

import Opetope as O
import OpetopesUtils
import Data.AVL.Set as S
import Data.SortedBag as MS
import Utils as U

%access export

public export
data ProdFace : Nat -> Type where
    Point : O.Opetope Z -> O.Opetope Z -> ProdFace Z
    Arrow : O.Opetope k1 -> O.Opetope k2 -> ProdFace Z -> ProdFace Z
            -> ProdFace (S Z)
    Face : O.Opetope k1 -> O.Opetope k2 -> List (ProdFace (S m))
           -> ProdFace (S m) -> ProdFace (S (S m))

public export
flip : ProdFace n -> ProdFace n
flip (Point p q) = Point q p
flip (Arrow p q d c) = Arrow q p (flip d) (flip c)
flip (Face p q d c) = Face q p (map flip d) (flip c)


export
dom : (ProdFace (S n)) -> List (ProdFace n)
dom (Arrow _ _ d _) = [d]
dom (Face _ _ d _) = d

export
cod : (ProdFace (S n)) -> (ProdFace n)
cod (Arrow _ _ _ c) = c
cod (Face _ _ _ c) = c

public export
dim : {n: Nat} -> (ProdFace n) -> Nat
dim {n} _ = n


helper_dim: {n: Nat} -> (ProdFace n) -> Nat
helper_dim (Point _ _) = Z
helper_dim (Arrow _ _ _ _) = (S Z)
helper_dim (Face _ _ _ c) = S (dim c)

lemma_dim_eq_helper_dim: {n: Nat} -> (g: ProdFace n) -> dim g = helper_dim g
lemma_dim_eq_helper_dim {n = Z} (Point x y) = Refl
lemma_dim_eq_helper_dim {n = (S Z)} (Arrow x y z w) = Refl
lemma_dim_eq_helper_dim {n = (S (S m))} (Face x y xs z) = Refl

export
total
dim_p1 : ProdFace n -> Nat
dim_p1 (Point p _) = dim p
dim_p1 (Arrow p _ _ _) = dim p
dim_p1 (Face p _ _ _) = dim p


export
total
dim_p2 : ProdFace n -> Nat
dim_p2 (Point _ q) = dim q
dim_p2 (Arrow _ q _ _) = dim q
dim_p2 (Face _ q _ _) = dim q

total
lemma_of_dim_op : {n: Nat} -> (op: Opetope n) -> (n = (dim op))
lemma_of_dim_op {n = Z} (Point x) = Refl
lemma_of_dim_op {n = (S Z)} (Arrow x y z) = Refl
lemma_of_dim_op {n = (S (S k))} (Face x xs y) = Refl

total
lemma_of_dim_face : {n: Nat} -> (g: ProdFace n) -> (n = (dim g))
lemma_of_dim_face {n = Z} (Point x y) = Refl
lemma_of_dim_face {n = (S Z)} (Arrow x y z w) = Refl
lemma_of_dim_face {n = (S (S m))} (Face x y xs z) = Refl


export
p1 : (g: ProdFace n) -> Opetope (dim_p1 g)
p1 g = case g of
     (Point p _) => replace (lemma_of_dim_op p) p
     (Arrow p _ _ _) => replace (lemma_of_dim_op p) p
     (Face p _ _ _) => replace (lemma_of_dim_op p) p

export
p2 : (g: ProdFace n) -> Opetope (dim_p2 g)
p2 g = case g of
     (Point _ q) => replace (lemma_of_dim_op q) q
     (Arrow _ q _ _) => replace (lemma_of_dim_op q) q
     (Face _ q _ _) => replace (lemma_of_dim_op q) q
```

```
mutual
    public export
    Eq (ProdFace n) where
        (Point p q) == (Point p' q')           = (p, q) == (p', q')
        (Arrow p q d c) == (Arrow p' q' d' c') = O.eq p p' &&
                                                 O.eq q q' &&
                                                 (c, d) == (c', d')
        (Face p q d c) == (Face p' q' d' c')   = O.eq p p' &&
                                                 O.eq q q' &&
                                                 (c, sort d) == (c', sort d')

    public export
    Eq (ProdFace n) => Ord (ProdFace n) where
        compare (Point p q) (Point p' q')           = compare (p, q) (p', q')
        compare (Arrow p q d c) (Arrow p' q' d' c') = lexi_order (p, q, d, c) (p', q', d', c')
        compare (Face p q d c) (Face p' q' d' c')   = lexi_order (p, q, sort d, c) (p', q', sort d', c')

    lexi_order : (Ord a) => (Opetope n1, Opetope n2, a) -> (Opetope k1, Opetope k2, a) -> Ordering
    lexi_order (a1, a2, x) (b1, b2, y) = case (O.comp a1 b1, O.comp a2 b2, compare x y) of
        (LT, _, _) => LT
        (EQ, LT, _) => LT
        (EQ, EQ, LT) => LT
        (EQ, EQ, EQ) => EQ
        _ => GT


public export
Show (ProdFace n) where
    show (Point p q) = show p ++ show q
    -- show (Arrow p q d c) = (show $ 1) ++ "!(" ++ show [d] ++ " -> " ++ show c ++ ")"
    -- show (Face p q d c) = (show $ dim c + 1) ++ "!(" ++ show d ++ "->" ++ show c ++ ")"
    show (Arrow p q d c) = (show $ 1) ++ "!(" ++ show ((p, q)) ++ ": " ++ show [d] ++ " -> " ++ show c ++ ")"
    show (Face p q d c) = (show $ dim c + 1) ++ "!(" ++ show ((p, q)) ++ ": " ++ show d ++ "->" ++ show c ++ ")"


name_of_op : O.Opetope k -> O.Opetope l -> String
name_of_op p q = show (p, q)


embed : {n:Nat} -> (g: ProdFace n) -> O.Opetope (dim g)
embed {n} op = case op of
        (Point p q) => O.Point (name_of_op p q)
        (Arrow p q d c) => O.Arrow (name_of_op p q) (embed d) (embed c)
        (Face p q d c) =>  O.Face (name_of_op p q) (map embed d) (embed c)

public export
match : ProdFace n -> Bool
match op = case op of
    (Point _ _) => True
    (Arrow _ _ _ _) => True
    (Face _ _ _ _) => O.match (embed op)


all_eq : List (Opetope k) -> Opetope l -> Bool
all_eq ls op = U.and_ (map (\x => O.eq x op) ls)

transform_n_k : {n: Nat} -> (k: Nat) -> Opetope n -> Maybe (Opetope k)
transform_n_k k op = case decEq (dim op) k of
        Yes prf => Just (replace prf op)
        No _ => Nothing


all_eq_lsts : List (Opetope k1) -> List (Opetope k2) -> Bool
all_eq_lsts (x::xs) (y::ys) = case decEq (dim x) (dim y) of
    Yes prf => (MS.fromList (replace prf (x::xs))) == (MS.fromList (y::ys))
    No _ => False
all_eq_lsts [] [] = True
all_eq_lsts _ _ = False

contracted_eq : {k1: Nat} -> {k2: Nat} -> List (Opetope k1) -> Opetope k1 -> Opetope k2 -> Bool
contracted_eq {k1=Z} {k2=Z} ins out op = all_eq (out::ins) op
contracted_eq {k1=(S l1)} {k2=(S l2)} ins out op = (O.eq (cod op) out && all_eq_lsts ins (dom op)) ||
                              (all_eq ins out && contracted_eq (dom out) (cod out) op)
contracted_eq {k1=Z} {k2=(S l2)} ins out op = (O.eq (cod op) out && all_eq_lsts ins (dom op))
contracted_eq {k1=(S l1)} {k2=Z} ins out op = (all_eq ins out && contracted_eq (dom out) (cod out) op)

deep_p1_m : {n: Nat} -> (g: ProdFace n) -> Bool
deep_p1_m (Point p _) = True
deep_p1_m (Arrow p _ d c) = case p of
    (O.Point _) => O.eq (p1 d) (p1 c) && O.eq p (p1 c)
    (O.Arrow _ st fn) => O.eq (p1 d) st && O.eq (p1 c) fn
    _ => False
deep_p1_m {n = (S (S m))} (Face p _ d c) =
        (U.and_ (map deep_p1_m d)) && (deep_p1_m c) && (contracted_eq ins out p)
    where
        out : Opetope (dim_p1 c)
        out = p1 c
        ins : List (Opetope (dim_p1 c))
        ins = catMaybes $ map (\x => transform_n_k (dim $ p1 c) (p1 x)) d


deep_p2_m : {n: Nat} -> (g: ProdFace n) -> Bool
deep_p2_m (Point _ q) = True
deep_p2_m (Arrow _ q d c) = case q of
```

```
    (O.Point _) => O.eq (p2 d) (p2 c) && O.eq q (p2 c)
    (O.Arrow _ st fn) => O.eq (p2 d) st && O.eq (p2 c) fn
    _ => False
deep_p2_m {n = (S (S m))} (Face _ q d c) =
        (U.and_ (map deep_p2_m d)) && (deep_p2_m c) && (contracted_eq ins out q)
    where
        out : Opetope (dim_p2 c)
        out = p2 c
        ins : List (Opetope (dim_p2 c))
        ins = catMaybes $ map (\x => transform_n_k (dim $ p2 c) (p2 x)) d


export
is_valid : {n: Nat} -> ProdFace n -> Bool
is_valid {n} g = match g && deep_p1_m g && deep_p2_m g


export
from_point_and_point : O.Opetope Z -> O.Opetope Z -> ProdFace Z
from_point_and_point p1 p2 = Point p1 p2

export
from_arrow_and_point : O.Opetope (S Z) -> O.Opetope Z -> ProdFace (S Z)
from_arrow_and_point arr pt = let (O.Arrow _ d c) = arr in
    Arrow arr pt (Point d pt) (Point c pt)


export
from_point_and_arrow :  O.Opetope Z -> O.Opetope (S Z) -> ProdFace (S Z)
from_point_and_arrow pt arr = flip (from_arrow_and_point arr pt)

export
from_arrow_and_arrow : O.Opetope (S Z) -> O.Opetope (S Z) -> ProdFace (S Z)
from_arrow_and_arrow arr1 arr2 =
    let (O.Arrow _ d1 c1) = arr1
        (O.Arrow _ d2 c2) = arr2 in
            Arrow arr1 arr2 (Point d1 d2) (Point c1 c2)

public export
FSet : Nat -> Type
FSet n = S.Set (ProdFace n)

export
singleton : {n: Nat} -> ProdFace n -> FSet n
singleton {n} op = S.insert op S.empty

export
unions : (Foldable t) => t (FSet n) -> FSet n
unions os = foldr S.union S.empty os
```

Listing B.4: FacesUtils.idr

```
module FacesUtils

import Face as F
import Data.AVL.Set as S

%access public export

FMap : Type
FMap = (n: Nat) -> FSet n

empty: FMap
empty n = S.empty

fromFSet : {n: Nat} -> (FSet n) -> FMap
fromFSet {n} f = \k => case decEq n k of
   Yes prf => replace prf f
   No _ => S.empty

singleton : {n: Nat} -> (ProdFace n) -> FMap
singleton {n} x = \k => case decEq n k of
   Yes prf => F.singleton (replace prf x)
   No _ => S.empty

get : (n:Nat) -> FMap -> FSet n
get n om = om n

union : FMap -> FMap -> FMap
union om1 om2 = \n => S.union (get n om1) (get n om2)

unions : (List FMap) -> FMap
unions [] = empty
unions (x::xs) = x `union` (unions xs)

fromList : List (ProdFace k) -> FMap
fromList [] = empty
fromList (x::xs) = (singleton x) `union` (fromList xs)

Show FMap where
    show t = show' t 0
        where
            show' : FMap -> Nat -> String
            show' t n = if (p == S.empty) && n > 2 then " \n "
                        else (show n ++ "/" ++ show (size p) ++ " :: " ++
                             (show p) ++ " \n " ++ (show' t (n + 1)))
                where
                    p : FSet n
                    p = t n

dmap: Functor f => (func : a -> b) -> f a -> f (Lazy b)
dmap func it = map (Delay . func) it
```

Listing B.5: Utils.idr

```
module Utils

%access public export
Show Ordering where
    show LT = "LT"
    show GT = "GT"
    show EQ = "EQ"

Ord Ordering where
    compare LT EQ = LT
    compare LT GT = LT
    compare EQ GT = LT

    compare EQ EQ = EQ
    compare LT LT = EQ
    compare GT GT = EQ

    compare EQ LT = GT
    compare GT LT = GT
    compare GT EQ = GT


dmap: Functor f => (func : a -> b) -> f a -> f (Lazy b)
dmap func it = map (Delay . func) it

and_ : List Bool -> Bool
and_ [] = True
and_ (x::xs) = x && (and_ xs)

cart_prod_with : (a -> b -> c) -> List a -> List b -> List c
cart_prod_with f as bs = [f a b | a <- as, b <- bs]

natFromTo : Nat -> Nat -> List Nat
natFromTo b e = if b <= e then natEnumFromTo b e else []
```

Listing B.6: Product.idr

```
module Product

import Opetope as O
import OpetopesUtils as OU
import Face as F
import FacesUtils as FU

import Data.AVL.Set as S
import Data.SortedBag as MS

import Utils as U

%access public export

dfs : FSet n -> FSet (S n) -> FSet (S n) -> ProdFace (S n) -> Opetope k1
      -> Opetope k2 -> FSet (S (S n))
dfs ins used building_blocks target_out p q =
    let f = F.Face p q (S.toList used) target_out in
    if F.is_valid f
        then F.singleton f
        else F.unions [(dfs new_ins
                            new_used
                            building_blocks
                            target_out
                            p q) | b <- S.toList building_blocks,
                                   i <- S.toList ins,
                                   not (S.contains b used),
                                   S.contains (cod b) (holes_to_fill used target_out),
                                   let new_ins = (ins `S.union` (S.fromList (F.dom b))),
                                   let new_used = (b `S.insert` used)]
    where
        holes_to_fill : FSet (S n) -> ProdFace (S n) -> FSet n
        holes_to_fill s t = (S.insert (F.cod t) (S.fromList (concat $ map F.dom (S.toList s)))
                           `S.difference`
                           (S.fromList $ map F.cod (S.toList s)))
possible_faces : F.ProdFace (S n) -> List (F.ProdFace (S n)) -> Opetope k1 -> Opetope k2 -> FSet (S (S n))
possible_faces op building_blocks p q =
    dfs (F.singleton (F.cod op)) S.empty (S.fromList building_blocks) op p q


mul_0k : O.Opetope Z -> O.Opetope k -> F.ProdFace k
mul_0k p q = case q of
    (O.Point _) => F.Point p q
    (O.Arrow _ d c) => F.Arrow p q (F.Point p d) (F.Point p c)
    (O.Face _ d c) => F.Face p q (map (\s => mul_0k p s) d) (mul_0k p c)

mul_k0 : O.Opetope k -> O.Opetope Z -> F.ProdFace k
mul_k0 p q = case p of
    (O.Point _) => F.Point p q
    (O.Arrow _ d c) => F.Arrow p q (F.Point d q) (F.Point c q)
    (O.Face _ d c) => F.Face p q (map (\s => mul_k0 s q) d) (mul_k0 c q)


base_case_0k : {k: Nat} -> O.Opetope Z -> O.Opetope k -> FU.FMap
base_case_0k {k} p q = FU.unions [FU.fromList (map (\s => mul_0k p s)
    (MS.toList $ (OU.subopetopes q n))) | n <- natRange (S k)]

base_case_k0 : {k: Nat} -> O.Opetope k -> O.Opetope Z -> FU.FMap
base_case_k0 {k} p q = FU.unions [FU.fromList (map (\s => mul_k0 s q)
    (MS.toList $ (OU.subopetopes p n))) | n <- natRange (S k)]


getIf : List a -> Bool -> List a
getIf l b = if b then l else []


big_product : Nat -> FU.FMap -> O.Opetope (S k1) -> O.Opetope (S k2) -> (FU.FMap, Nat)
big_product (S (S k)) curr_faces p q =
        if new_faces == S.empty && k > 1 then
            (curr_faces, (S (S k)))
        else
            big_product (S (S (S k))) (FU.union (FU.fromFSet new_faces) curr_faces) p q
    where
        maxd : Nat
        maxd = (maximum (dim p) (dim q))
        build_new_opetopes : F.ProdFace (S k) -> FSet (S (S k))
        build_new_opetopes op = possible_faces op building_blocks p q where
            building_blocks : List (F.ProdFace (S k))
            building_blocks = [s | s <- S.toList $ curr_faces (S k),
                                   op /= s]
        possible_codomains : List (F.ProdFace (S k))
        possible_codomains = [s | s <- faces,
                                      O.eq (p1 s) p,
                                      O.eq (p2 s) q] ++
                             (getIf [s | s <- faces,
                                             O.eq (p1 s) (cod p),
                                             O.eq (p2 s) (cod q)]
                                    ((S (S k)) == maxd)) ++
                             (getIf [s | s <- faces,
                                             O.eq (p1 s) p,
                                             O.eq (p2 s) (cod q)]
                                    ((S (S k)) == maxd && (dim p) < (dim q))) ++
```

```
                            (getIf [s | s <- faces,
                                            O.eq (p1 s) (cod p),
                                            O.eq (p2 s) q]
                                   ((S (S k)) == maxd && (dim p) > (dim q))) where
            faces : List (ProdFace (S k))
            faces = S.toList $ curr_faces (S k)

        new_faces : F.FSet (S (S k))
        new_faces = F.unions $ map build_new_opetopes possible_codomains


mutual
    small_faces : {k1: Nat} -> {k2: Nat} -> O.Opetope (S k1) -> O.Opetope (S k2) -> FU.FMap
    small_faces {k1} {k2} p q =
        let pt_pt = make from_point_and_point (subs p Z) (subs q Z)
            pt_ar = make from_point_and_arrow (subs p Z) (subs q (S Z))
            ar_pt = make from_arrow_and_point (subs p (S Z)) (subs q Z)
            ar_ar = make from_arrow_and_arrow (subs p (S Z)) (subs q (S Z)) in
        FU.unions ([pt_pt, pt_ar, ar_pt, ar_ar] ++ small_products) where
            make : (a -> b -> ProdFace nk) -> List a -> List b -> FU.FMap
            make f l1 l2 = FU.fromList $ U.cart_prod_with f l1 l2
            subs : O.Opetope k -> (n: Nat) -> List (O.Opetope n)
            subs op n = MS.toList $ OU.subopetopes op n
            small_products : List (FU.FMap)
            small_products =  [fst (product r s) | n1 <- U.natFromTo Z (S k1),
                                            n2 <- U.natFromTo Z (S k2),
                                            r <- subs p n1,
                                            s <- subs q n2,
                                            n1 < (S k1) || n2 < (S k2),
                                            n1 > Z || n2 > Z]

    product : {k1: Nat} -> {k2: Nat} -> O.Opetope k1 -> O.Opetope k2 -> (FU.FMap, Nat)
    product {k1} {k2} p q = case (p, q) of
            ((Point _), _) => (base_case_0k p q, dim q)
            (_, (Point _)) => (base_case_k0 p q, dim p)
            (Arrow _ _ _, Arrow _ _ _) => (big_product (S (S Z)) (small_faces p q) p q)
            (Face _ _ _, Arrow _ _ _) => (big_product (maximum (dim p) (dim q)) (small_faces p q) p q)
            (Arrow _ _ _, Face _ _ _) => (big_product (maximum (dim p) (dim q)) (small_faces p q) p q)
            (Face _ _ _, Face _ _ _) => (big_product (maximum (dim p) (dim q)) (small_faces p q) p q)
```

Listing B.7: Main.idr

```
module Main

import Opetope as O
import OpetopesUtils
import Face as F
import Product as P
import FacesUtils
import Utils as U

a: Opetope Z
a = O.Point "a"
b: Opetope Z
b = O.Point "b"
e: Opetope Z
e = O.Point "e"
ab1: Opetope (S Z)
ab1 = O.Arrow "ab1" a b
ab2: Opetope (S Z)
ab2 = O.Arrow "ab2" a b
alpha: Opetope (S (S Z))
alpha = O.Face "alpha" [ab1] ab2

ab : Opetope (S Z)
ab = O.Arrow "ab" a b
be : Opetope (S Z)
be = O.Arrow "be" b e
ae : Opetope (S Z)
ae = O.Arrow "ae" a e

three : Opetope (S (S Z))
three = O.Face "three" [ab, be] ae

c: Opetope Z
c = O.Point "c"
d: Opetope Z
d = O.Point "d"
cd1: Opetope (S Z)
cd1 = O.Arrow "cd1" c d

pAC: F.ProdFace Z
pAC = F.Point a c
pAD: F.ProdFace Z
pAD = F.Point a d
pBC: F.ProdFace Z
pBC = F.Point b c
pBD: F.ProdFace Z
pBD = F.Point b d

aACAD: F.ProdFace (S Z)
aACAD = (F.Arrow a cd1 pAC pAD)
aADBD1: F.ProdFace (S Z)
aADBD1 = (F.Arrow ab1 d pAD pBD)
aADBD2: F.ProdFace (S Z)
aADBD2 = (F.Arrow ab2 d pAD pBD)
aACBD1: F.ProdFace (S Z)
aACBD1 = (F.Arrow ab1 cd1 pAC pBD)
aACBD2: F.ProdFace (S Z)
aACBD2 = (F.Arrow ab2 cd1 pAC pBD)

s1 : F.ProdFace 2
s1 = F.Face ab1 cd1 [aACAD, aADBD1] aACBD1
s2 : ProdFace 2
s2 = F.Face alpha cd1 [aACAD, aADBD1] aACBD2
s3 : ProdFace 2
s3 = F.Face alpha cd1 [aACBD1] aACBD2

sd : ProdFace 3
sd = F.Face alpha cd1 [s3, s1] s2

sw : ProdFace 2
sw = F.Face alpha cd1 [aADBD1] aADBD2
-- p : F.ProdFace (S (S Z))
-- p = F.Face ab1 cd1 [aACAD, aADBD] aACBD

op : String
op = show $ (P.product alpha alpha)

main : IO ()
main = putStrLn $ (show $ op)
```

Bibliography

[1] T. Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: `https://homotopytypetheory.org/book`, 2013.

[2] G. Friedman, "Survey article: An elementary illustrated introduction to simplicial sets," *Rocky Mountain J. Math.*, vol. 42, pp. 353–423, 04 2012.

[3] M. Zawadowski, "On positive face structures and positive-to-one computads," *ArXiv e-prints*, Aug. 2007.

[4] M. Zawadowski, "Positive Opetopes with Contractions form a Test Category," *ArXiv e-prints*, Dec. 2017.

[5] E. Finster, "Opetopes visualizations," 2018.

[6] J. Kock, A. Joyal, M. Batanin, and J.-F. Mascari, "Polynomial functors and opetopes," *Advances in Mathematics*, vol. 224, pp. 2690–2737, aug 2010.

[7] S. Szawiel and M. Zawadowski, "The web monoid and opetopic sets," *Journal of Pure and Applied Algebra*, vol. 217, no. 6, p. 1105–1140, 2013.