Distributional models of meaning

Semantical, grammatical, and moral ambiguity

Mario Román

27th March 2019

Contents

Introduction: a mathematical model

The distributional compositional categorical framework (DisCoCat) of [CSC10] is based on the fact that both the grammar and meaning can share the structure of a compact closed monoidal category. On the one hand, grammar can be expressed by pregroups [Lam97], which happen to be compact closed posets. On the other hand, in *distributional semantics*, the meanings are represented in the compact closed category of finite vector spaces. Once we accept some level abstraction, however, we are not limited anymore to vector spaces. Examples of models outside vector spaces include Gärdenfors' conceptual spaces [Gär04], as developed in [BCG+17], or categories of generalized relations, as in [CGL+18].

We will follow [Ash15] to briefly describe the framework in a manner that will be useful to our exposition. We can fix some types for our grammar, (e.g. $G = \{n, s\}$) and we take the free compact-closed category $\mathbf{G}$ over them: each element a will have some left adjoint a^l and some right adjoint a^r and we will have the following morphisms determining the dualities.

$$1 \to aa^l, \quad 1 \to a^r a, \quad aa^r \to 1, \quad a^l a \to 1.$$

We will be primarily using the category of relations $\mathbf{Rel}$ as our model of meaning. In our case, the categories $\mathbf{G}$ and $\mathbf{Rel}$ have a compact closed structure in common that will be preserved by any strong monoidal functor.

In this text, we will define mathematical meanings over some models and then we will use them to tell parts of the story of a play by Camus. We will also propose a technique for transforming grammatical ambiguity into semantic ambiguity. In practice, this technique will allow the *grammatical* parsing of a sentence to be affected by the *meaning* of the words. Because of their simplicity, we will make use of the compact closed category of relations $\mathbf{Rel}$ and its free enrichment over convex algebras $\mathbf{Rel}_D$. However, we think it would be easy to extend the same ideas to other categories of generalized relations as described in [CGL+18].

We finally propose an implementation of all our computations in the Haskell programming language [HHJW07]. The structure of this programming language will be specially suited for the kind of models we will work with. Moreover, the notation of the language can be useful to informally write down morphisms of $\mathbf{Rel}$. On the other hand, we have decided not to pursue the *distributional* part of the model: we will just assume that the meanings of the words have been given beforehand.

Experimental setting: Les Justes

Our experimental setting will be based on the 1949 play "*Les justes*" ("The Righteous" or "The just assassins") by Albert Camus [Cam49]. The justification for this choice is that (1) the play has a very limited set of characters, (2) we will have examples of ambiguity in the meaning, (3) these same examples will be useful for dealing with ambiguity on the grammar. We also will take one of the ideas from DisCoCirc framework of [Coe19] for representing an evolution of the characters; namely, that verbs will be processes instead of just states.

Characters and sentences

On our stage, we have a group of revolutionaries (Yanek, Dora, Stepan and Boris) plotting the assassination of the Grand Duke Alexandrovich. The first acts of the play revolve around a furious debate among the revolutionaries on whether and when their violence is morally justified. Two minor characters of the play but relevant to our modeling will be the Duke's nephew and the police officer Skouratov. We fix a set $\mathbf{Nouns}$ and then all the characters, with grammatical type n, can be seen as relations $1 \to \mathbf{Nouns}$, or, in other words, as subsets of a big space of nouns. We write this in Haskell notation as follows.

```haskell
data Nouns = Yanek | Dora | Duke | Stepan | Boris | Nephew | Skouratov
           | Poet | Revolutionary | Terrorist | Innocent |
           | Tsarist | Alive | Saviour
           | Life | Poetry | Chemistry | Propaganda | Bomb

yanek  = [Yanek, Poet, Revolutionary, Alive, Innocent]
stepan = [Stepan, Revolutionary, Alive]
dora   = [Dora, Revolutionary, Alive, Innocent]
duke   = [Duke, Tsarist, Alive]
```

Note that we have chosen to have both a basis element for each character (their *identity*, that never changes) and a description in terms of other basis elements (their *description*, that may change).

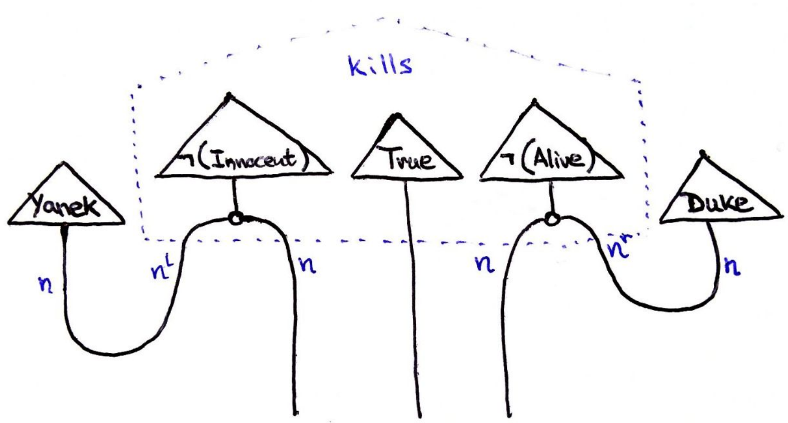
Example 1: semicartesian verb

Transitive verbs will have type $n^l s n^r$ in the Lambek grammar, meaning that they should be represented by relations $1 \to \mathbf{Nouns} \otimes \mathbf{Sentence} \otimes \mathbf{Nouns}$. If we also want our subjects and objects to evolve, as they do in the DisCoCirc framework developed in [Coe19], we can use relations of the form $\mathbf{Nouns} \otimes \mathbf{Nouns} \to$

Nouns $\otimes$ **Sentence** $\otimes$ **Nouns**. In this example, the verb *kill* is represented almost like an identity relation, with the exception that it blocks the possibility of being "*innocent*" anymore for the subject, and the possibility of being "*alive*" for the object. We have a sentence meaning plot ∈ **Sentence** that will be useful later.

$$\mathsf{kills} = (a, b) \mapsto (a, \mathsf{plot}, b) \quad \text{if } (a \neq \mathsf{Innocent}) \wedge (b \neq \mathsf{Alive}).$$

This can be seen then as an example of a **semicartesian verb,** as defined in [CLM18]. In order to write it down in a compact way, we will need some notion of negation. Finding some morphism that suitably represents negation is a problem in the models of meaning we are using, so we will use an operation outside the model to represent it. Given any relation $s\colon 1 \to A$ we take $(\neg s)\colon 1 \to A$ to be a relation given by the complement subset. This negation operation $(\neg)$ must not be interpreted as having any particular meaning on the framework (it is not a morphism, after all), but as a notational convention on top of it.



We have our first example of an evaluation here:

- *Yanek kills the Duke.*

It is translated to the Haskell implementation as follows. The implementation automatically finds and computes the necessary Lambek grammar reduction.

```
-- Definition of 'kills' as a semicartesian verb. We write
-- concatenation as (<>). The Lambek grammar type is given after
-- '@@'.
kills = (lnot Innocent <> cnst IsTrue <> rnot Alive)
```

```
  @@ [L N, N, S, N, R N]

-- Let's compare Yanek before and after the murder. We discard the two
-- wires we do not need in the second case. In the first case, Yanek
-- is innocent.
yanek
>> Yanek, Poet, Revolutionary, Alive, Innocent
-- However, after the murder, he stops being innocent.
(yanek <> kills <> duke) <> discard <> discard
>> Yanek, Poet, Revolutionary, Alive
```

Example 2: relative pronouns

At the first stages of the play, the moral differences between the protagonists are highlighted. They represent different moral positions regarding how the revolution and the greater good justifies their actions. We will study these differences using relative pronouns.

There are two possible interpretations of a sentence: (1) as asserting the truth of something or providing some new information to our model, or (2) as a query that can be answered with the information we have. The modelling of pronouns in [SCC14] seems to be better suited for the second kind of interpretation. On the other hand, Example 1 was working with the first interpretation in mind.

Central to this discussion is knowing who are the revolutionaries *combatting*; who are they plotting to murder. We also consider the verb *to be*, used to describe the characters; and two transitive verbs with the same meaning, *enjoy* and *like*. That is, we will have morphisms combat, likes, is : $1 \to \mathbf{Nouns} \otimes \mathbf{Sentence} \otimes \mathbf{Nouns}$. We write them as subsets in Haskell notation, together with their grammatical types.

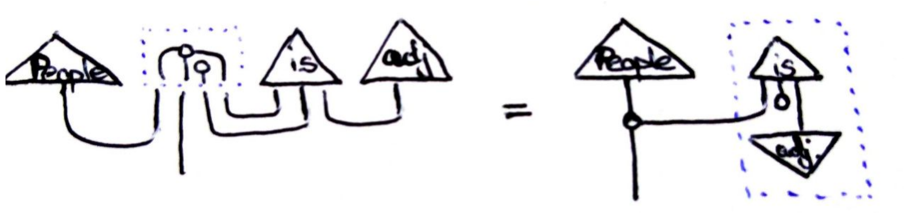
```
combat = Words (fromList
  [ [Yanek   , IsTrue , Duke]
  , [Dora    , IsTrue , Duke]
  , [Stepan  , IsTrue , Duke]
  , [Yanek   , IsTrue , Skouratov]
  , [Dora    , IsTrue , Skouratov]
  , [Stepan  , IsTrue , Skouratov]
  , [Skouratov , IsTrue , Yanek]
  , [Skouratov , IsTrue , Dora]
  , [Skouratov , IsTrue , Stepan]
  , [Stepan    , IsTrue , Nephew]
```

```
  ]) [L N , S , R N]

is = Words (fromList
  [ [Yanek, IsTrue, Revolutionary]
  , [Yanek, IsTrue, Poet]
  , [Dora, IsTrue, Revolutionary]
  , [Stepan, IsTrue, Revolutionary]
  , [Stepan, IsTrue, Terrorist]
  , [Boris, IsTrue, Revolutionary]
  , [Duke, IsTrue, Tsarist]
  , [Skouratov, IsTrue, Tsarist]
  , [Nephew, IsTrue, Innocent]
  ]) [L N, S, R N]

enjoy = Words (fromList
  [ [Yanek, IsTrue, Poetry]
  , [Yanek, IsTrue, Life]
  , [Dora, IsTrue, Chemistry]
  , [Dora, IsTrue, Life]
  , [Stepan, IsTrue, Propaganda]
  , [Boris, IsTrue, Propaganda]
  , [Yanek , IsTrue , Dora]
  , [Dora , IsTrue , Yanek]
  , [Stepan , IsTrue , Dora]
  ]) [L N, S, R N]

likes = enjoy
```

With these descriptions, we are ready to define new compound concepts. For instance, the *tsarists* and the *revolutionaries* are precisely those described by these basis terms.

```
-- Tsarists = People who is tsarist.
tsarists        = (people <> who <> is <> tsarist      ) M.@@ [N]
>> [Duke , Skouratov]

-- Revolutionaries = People who is revolutionary.
revolutionaries = (people <> who <> is <> revolutionary) M.@@ [N]
>> [Yanek , Dora , Stepan , Boris]
```

The diagram with the corresponding reduction shows that the constructions *is adjt.* could have been interpreted as *intersective adjectives* (as in [BCG+17]).



We can go into more interesting (and even nested) queries, as in the following code. Note, however, that the second one presents a problem to the implementation. It is not the same to say "*People who combat (people who combat tsarists)*" than it is to say "(*People who combat people) who combat tsarists*". Right now, we will choose to declare how we want the grammatical reduction to work, but we will be able to deal with these ambiguities in later sections.

```
-- People who combat tsarists.
(people <> who <> combat <> tsarists) M.@@ [N]
>> [Yanek , Dora , Stepan]

-- People who combat people who combat tsarists.
(people <> who <> combat <>
  (people <> who <> combat <> tsarists) M.@@ [N])
    M.@@ [N]
>> [Skouratov]
```

Let's throw in some more examples! Thanks to our implementation, we can let the computer figure out the wirings by itself without having to input them.

```
-- "Revolutionaries who enjoy life enjoy propaganda" evaluates to
-- false.
(revolutionaries <> who <> enjoy <> life <> enjoy <> propaganda) M.@@ [S]
>> [] of grammar type [S]

-- "Yanek likes revolutionaries who enjoy poetry or chemistry"
-- evaluates to true. The meaning of "or" is a relation that gives
-- the union of the basis elements.
or = [ [a,a,b] ++ [a,b,b] | a <- universe , b <- universe]
(yanek' <> likes <> revolutionaries <> who <> enjoy
        <> poetry <> or <> chemistry) M.@@ [S]
>> [[IsTrue]] of grammar type [S]

-- "Revolutionaries that combat people who is innocent are terrorists"
-- Evaluates to true. There is some grammatical ambiguity that we
-- solve.
that = who
```

```
are = is
(revolutionaries <> who <> combat <> people <> that <> is <> innocent
                 <> are <> terrorist) M.@@ [S]
>> [[IsTrue]] of grammar type [S]
```

Semantic (and moral) ambiguity

The meaning category

In order to work with ambiguity of the grammar, we first need models that allow for ambiguity of the meaning to compare how the multiple interpretations of the grammar affect it. We choose one of the models suggested in [Mar17]. The model allows for probabilistic mixtures to be encoded in the hom-sets.

Definition 1. We define $\mathbf{Rel}_D$ as the free convex algebra enriched category over the category $\mathbf{Rel}$ of relations.

Let $D\colon \mathbf{Sets} \to \mathbf{Sets}$ be the finite distribution monad. We are saying that given two sets $a, b \in \mathbf{Sets}$, the morphisms between them in this category are given by the free convex algebra over the set of relations between them. That is, $\mathbf{Rel}_D(a, b) = D(\mathbf{Rel}(a, b))$. Using this category as our category of meaning spaces is justified by Theorem 5.14 in [Mar17].

Proposition 2. *$\mathbf{Rel}_D$ is a compact closed category. More generally, if $\mathbf{C}$ is a compact closed category, then $\mathbf{C}_D$ is a compact closed category.*

Meaning in dispute

During the play, Yanek and the rest of the protagonists confront a moral dilemma: is their revolution enough justification for their crimes? are they saviours of the people or simply terrorists? The meaning of being *revolutionaries* is something that not everyone agrees on; and we will model this as an ambiguity. Tsarists think that the revolutionaries are terrorists, while the revolutionaries think of themselves as saviours. Everyone will agree in drawing the line between protectors and terrorists on killing innocent people, but the question is which scenarios would imply the meaning of *revolutionaries* to be unambiguous. We take the definition of *revolutionary* to be an uniform mixture in $\mathbf{Rel}_D$ between *saviour* and *terrorist.*

$$\begin{aligned}\mathsf{revolutionary} &= 0.5\,\{\mathrm{Revolutionary}, \mathrm{Saviour}\} \\ &\quad + 0.5\,\{\mathrm{Revolutionary}, \mathrm{Terrorist}\}\end{aligned}$$

On the first act, Yanek tries to kill the Grand Duke. Yanek decides not to detonate the bomb when he discovers that the Duke is accompanied by his nephew. To understand the moral choice Yanek makes, we will consider two scenarios. The ambiguity on the identity of our protagonist is kept in one of the cases but not in the other.

Example 3: first scenario

Our first scenario is the following one.

- *Yanek is a revolutionary.*
- *He kills the Duke.*
- *Is he a terrorist?*

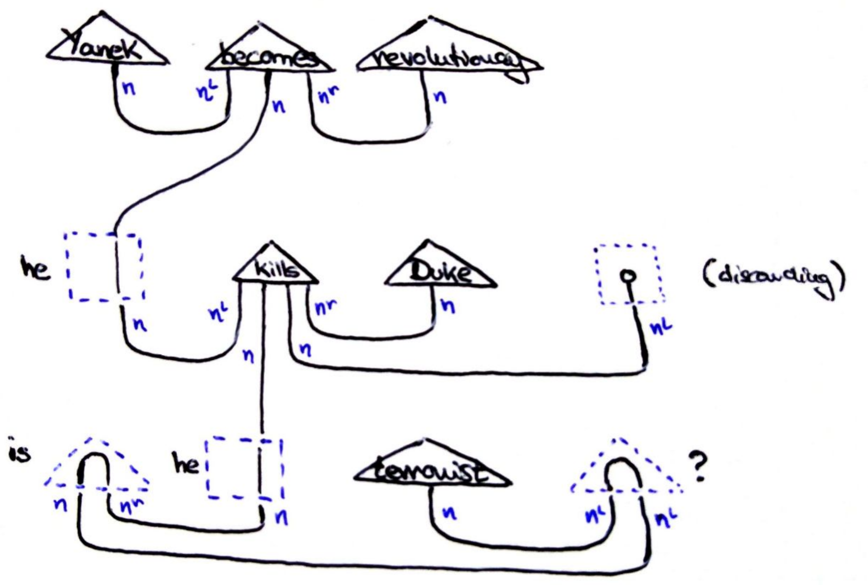
Here we take the verb *is* to mean *becomes*. We have a morphism $\mathsf{becomes}\colon 1 \to \mathbf{Nouns} \otimes \mathbf{Nouns} \otimes \mathbf{Nouns}$ that works using the basis noun Alive to add some adjectives to a character. Formally,

$$\mathsf{becomes} = \{(a, a, b) \mid a, b \in \mathbf{Nouns}\} \cup \{(\mathsf{Alive}, b, b) \mid b \in \mathbf{Nouns}\}\,.$$

Or, in Haskell notation.

```
becomes = Words (fromList $
    [ [ a ,     a , b ] | a <- universe , b <- universe ] ++
    [ [ Alive , b , b ] | b <- universe ])
  [L N , N , R N]
```

With a suitable interpretation of the functional words and considering the pronouns on a sentece as placeholders for the wires dangling from the previous sentences, we get the following diagram of the whole scene.



And our implementation gives back a scalar represented as a mixture of the two possible relations $1 \to 1$. Under the definition of terrorist, and considering that the Duke is causing the suffering of the Russian people (not innocent), it is still not clear that Yanek has become a terrorist in the eyes of everyone.

```
-- We set up the scenario as concatenating the three sentences.
scenario = (sentence1 <> sentence2 <> sentence3) M.@@ []
  where
    sentence1 = (yanek <> becomes <> revolutionary) M.@@ [N]
    sentence2 = (he <> kills <> duke <> discarding) M.@@ [L N , N]
    sentence3 = (is <> he <> terrorist <> (?)) M.@@ [L N]

-- We still get a mixture.
>> [[]] of grammar type [] with p=0.5
>> []   of grammar type [] with p=0.5
```

Example 4: second scenario

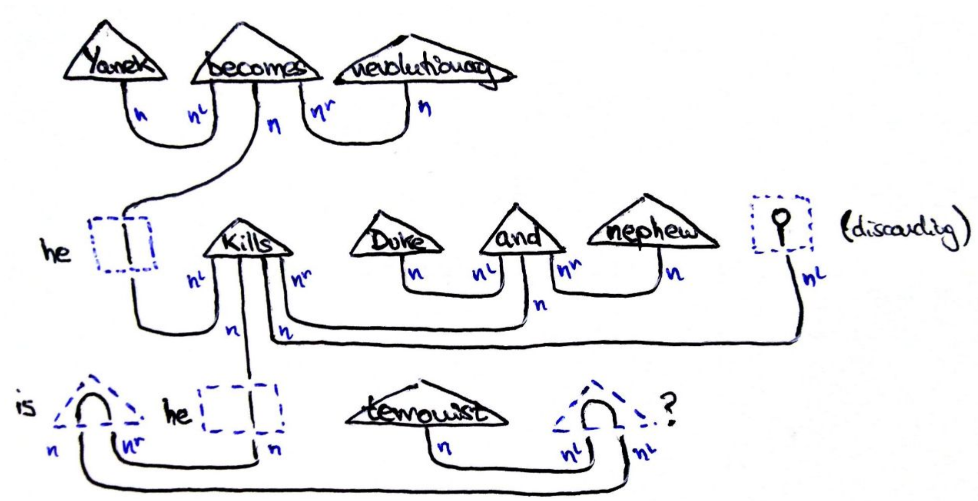
Our second scenario is:

- Yanek is a revolutionary.
- Yanek kills the duke *and his nephew*.
- Is Yanek a terrorist?

In this case, the word *"and"* is especially important. It could be argued that in some cases, an intersection of both nouns (a spider) would be the most suitable representation of *and*; but in this case, we want the pair to have the union of all the characteristics of its components. That means we will choose to define $\mathsf{and}\colon \mathbf{Nouns} \otimes \mathbf{Nouns} \to \mathbf{Nouns}$ as follows

$$\mathsf{and} = ((a,b) \mapsto a) \cup ((a,b) \mapsto b).$$

Incidentally, this is the same representation we used for "or" in Example 2. Again using a suitable representation of all the other functional words, we get the following diagram.



Now the results on the modified scenario are clear. Yanek ideals would not allow him to kill innocent people without considering himself a terrorist, and then the ambiguity would not exist anymore.

```
-- We set up the modified scenario.
scenario = (sentence1 <> sentence2 <> sentence3) M.@@ []
  where
    sentence1 = (yanek <> becomes <> revolutionary) M.@@ [N]
    both      = (duke <> and <> nephew) M.@@ [N]
    sentence2 = (he <> kills <> both <> discarding) M.@@ [L N , N]
    sentence3 = (is <> he <> terrorist <> (?)) M.@@ [L N]

-- We do not get a mixture anymore.
>> [[]] of grammar type [] with p=1
```

Some ideas on grammar ambiguity

The theory

In this section we will propose a technique to deal with grammatical ambiguity. When we use the DisCoCat framework, the grammar affects how the meanings of the different components of a sentence compose to create a meaning for the whole sentence. In this sense, *the grammar informs the meaning*. However, it seems much more difficult to devise a way in which the *meaning informs the grammar*. That is, the interactions in the meaning space should help us understand what the intended grammatical parsing was. Our proposed technique will use the meanings to resolve grammatical ambiguity.

We start with the basic definitions. Let $\mathbf{M}$ be some compact-closed category of meaning spaces (e.g. $\mathbf{FHilb}$ or $\mathbf{Rel}$). Let G be the the set of types (e.g. $\{n, s\}$) that determine a free compact-closed category $\mathbf{G}$ induced by their pregroup algebra.

Proposition 3. *A strong monoidal functor $\phi\colon \mathbf{G} \to \mathbf{M}$ must preserve the compact-closed structure. See [KSPC14].*

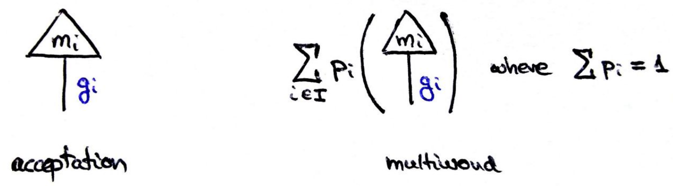
As $\mathbf{G}$ is freely generated, if we want to define one of these functors, it suffices to choose some meaning space for each one of the generating types using a function $[-]\colon G \to \mathrm{obj}(\mathbf{M})$.

The first thing we need is to deal with multiple meanings and grammars for the words at the same time. We will again make use of finite distributions; but this time, we want to mix different grammatical types.

Definition 4. An **acceptation** w is composed of a grammatical type $g \in \mathrm{obj}(\mathbf{G})$ and some state in the category of meaning spaces with output type given by the interpretation of that grammatical type, $m \in \mathbf{M}(I, [g_i])$.

Definition 5. A **multiword** w is given by the formal convex sum of multiple *acceptations* indexed by some finite set, which we call w_i for each $i \in I$. Each acceptation has then an associated probability p_i. In other words, a multiword is an element of the set of finite distributions over all possible acceptations.

For these, we introduce a notation that uses formal sums for multiwords and labels the wires not with the meaning space but with their grammatical type.



We also could write, in 1-dimensional notation, $\sum_{i \in I} p_i(m_i, g_i)$.

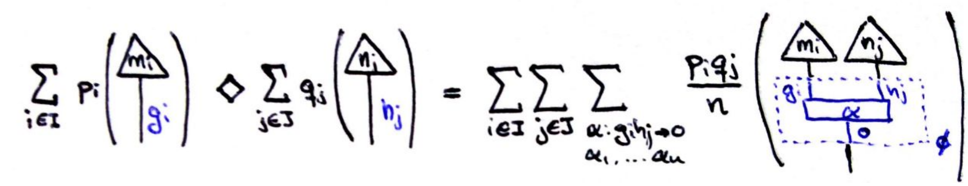
Next, we want to consider all possible grammatical reductions. We cannot consider *all* possible wirings of the words in a compact closed category. It would be infeasible in practice to work with all of them, and many would be simply adding unnecessary complexity. An observation about the DisCoCat framework is that (1) usually, wires do not need to cross, the category does not even need to be braided, and (2) we do not find caps on grammar reductions. This means we can focus on cups between contiguous tensored objects and on identities. We call *reductions* to wirings made up of these.

Definition 6. A **reduction** is a morphism in the category $\mathbf{G}$ that is generated by the composition and tensoring of both identities and cups of the form $g \otimes g^l \to 1$ and $g^r \otimes g \to 1$.

The following are examples and non-examples of reductions.

We finally introduce a meta-operation called **concatenation** that composes the meaning and grammar of multiple words, each one with multiple acceptations. The final result is again a formal sum of multiple meanings paired with different grammatical types.

Definition 7. The **concatenation** of two multiwords a and b will be written as $a \diamond b$ and it is defined as follows. Given two grammatical types g_i and h_j, we have α ranging over n possible reductions from these types. Note that $\phi\colon \mathbf{G} \to \mathbf{M}$ is a functor from grammar to meaning we defined earlier.



Or, in 1-dimensional notation.

$$\left(\sum_i p_i(m_i, g_i)\right) \diamond \left(\sum_j q_j(n_j, h_j)\right) = \sum_i \sum_j \sum_{\alpha:\ g_i h_j \to o} \frac{p_i q_j}{n} \left((m_i \otimes n_j) \circ \phi(\alpha), o\right)$$

Once we start composing multiple words, we can get a large formal sum with many different possible grammatical types. We are only interested in those that match our desired output, so we can *select* at the end of this process a particular grammatical type and discard all the others.

Definition 8. The **selection** operation (ดด) takes a multiword and a grammatical type and returns a new multiword that contains only the acceptations with that grammatical type. The probabilities are normalized to ensure that they add up to 1 again. In the case where there is no acceptation with the desired grammatical type, the function can return the empty word by convention.

In summary, we allow words to be a formal probabilistic mixture with different meanings and grammars. We concatenate them allowing all possible grammatical reductions to coexist, and finally we select only these that match the desired grammatical type.

Example 5: 'with' and the prepositional phrase attachment problem

In this section, we put the previous ideas on grammatical ambiguity into practice. Our example will concern the *prepositional phrase attachment* problem. An overview of the techniques that have been used to tackle it, and also an application of the compositional distributional framework can be found in [Del].

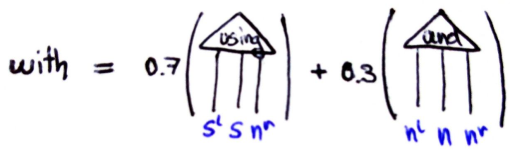
The word *with* has many different common acceptations. Moreover, it has different grammatical types on each one of these.

with. [CU]

1. Using something. *Join the pieces with glue.* ($s^l s n^r$)

2. Accompanied by. *Mix the butter with the sugar.* ($n^l s n^r$)

Let us assume that we can count the occurrences of each one of these on a reasonably big corpus of text, and that we find that, the first acceptation is used 70% of the time, whereas the second one is used only 30% of the time. We model it as a multiword with two acceptations, one corresponding to a verb *using* and the other corresponding to the already discussed *and* relation.



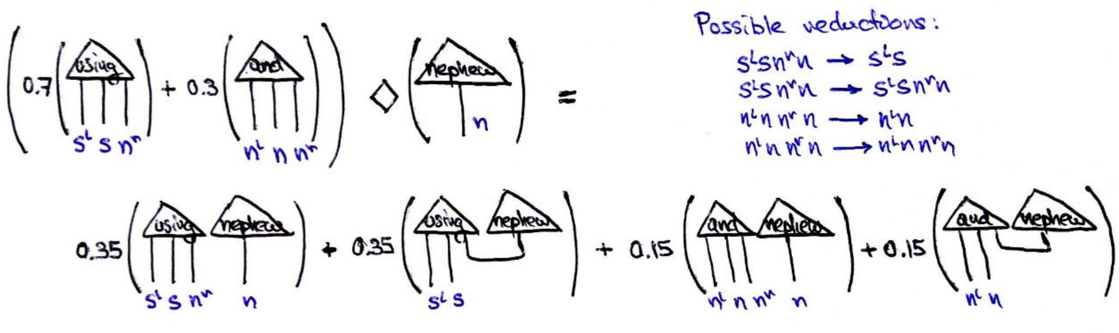
In our implementation this looks as follows. We include a new possible basis element for the sentence space describing a plot that can be carried with a tool. We then define a verb *using* that is quite limited but that suffices for our use case. We use the previous definition of *and.*

```
-- A plot can be carried using a bomb.
using = Words (fromList $
    [ [ Plot , Plot , Bomb ]
    ]) [L S , S , R N]

with :: M.Multiword Rel
with =  [ (using , 0.7) , (and , 0.3) ]
```

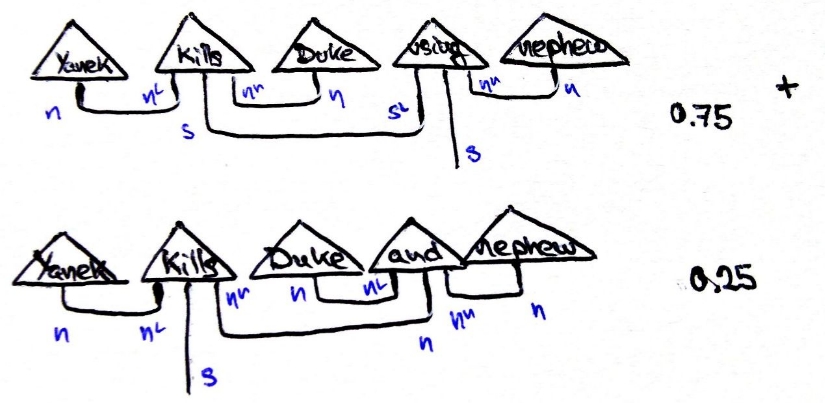
Now all the possible reductions on two sentences can be computed, and the ambiguity can be transferred from the grammar to the meaning. We want to compute two sentences that are examples of this behaviour.

- *Yanek kills the duke with his nephew.*
- *Yanek kills the duke with a bomb.*

As an example, we will start computing by hand part of the meaning of the first sentence. In this case, *"with his nephew"* as follows.



If we go through the whole sentence applying concatenations and then we select only these ones with the required grammatical type (in this case, s), we will get the following multiword.



But we precisely have an implementation to avoid doing these computations by hand. The two examples look as follows.

```
-- The first acceptation here can be rejected because it is empty.
-- This corresponds to the case where "and" is the correct
-- acceptation.
(yanek <> attacks <> duke <> with <> nephew)  M.@@ [S]
>> []       of grammar type [S] with p=0.756
>> [[Plot]] of grammar type [S] with p=0.243

-- Now both interpretations survive because our "and" word does not
-- complain if it has to mix a human and a bomb. However, it says that
-- the first interpretation ("using") is more plausible.
(yanek <> attacks <> duke <> with <> bomb)  M.@@ [S]
>> [[Plot]] of grammar type [S] with p=0.756
>> [[Plot]] of grammar type [S] with p=0.243
```

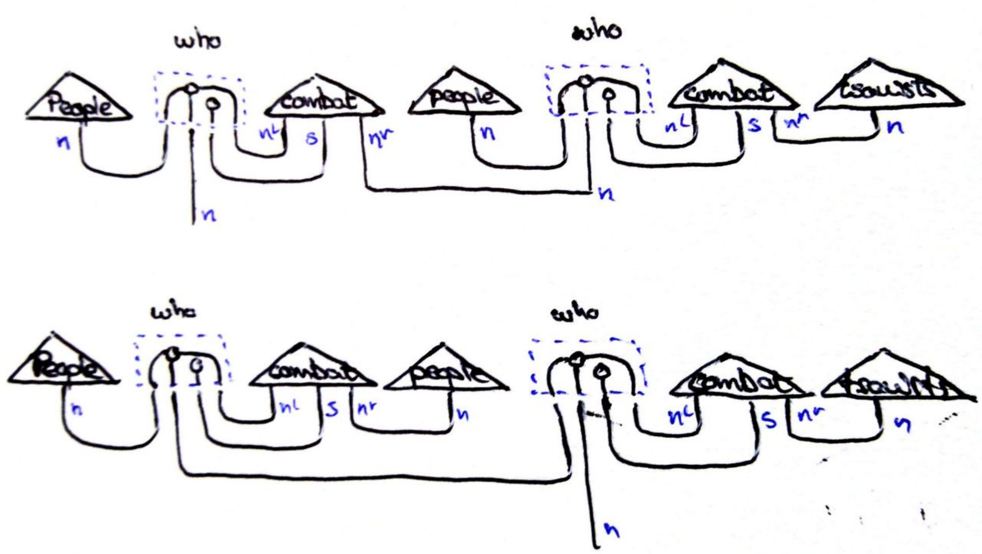
These definitions turn the first of the two summands into the empty relation, whereas they they turn the second into a meaningful sentence. That can be used to conclude that the correct interpretation for the first sentence was the one using $n^l n n^r$ as the grammatical type of *with*. We can also see how in the second sentence, the grammatical type $s^l s n^r$ is more likely.

Example 6: how plausible is each reduction

This technique solves the problem of grammatical ambiguities we encountered when dealing with nested relative pronouns. A more sophisticated solution could assign different probability weights to different reductions depending on how frequent these are in a real corpus of text, but we obtain interesting results simply by taking an uniform probability distribution each time we encounter multiple possible reductions.

```
(people <> who <> combat <> people <> who <> combat <> tsarists) M.@@ [N]
>> [[Yanek],[Dora],[Stepan]] of grammar type [N] with p=0.4285714
>> [[Skouratov]]             of grammar type [N] with p=0.5714285
```

This suggests that the second reading (the one we choose in our first approach to this sentence) is less convoluted than the second. In any case, we developed this metric only after developing and testing the model; it is non-associative, and it is unclear if it could be of any potential use. We show both possible reductions below, the first one got 0.57, whereas the second one got 0.43 under this metric.



We had another sentence with a similar problem whose resolution we present as a second example.

```
-- "Revolutionaries who combat people that is innocent are terrorists"
(revolutionaries <> who <> combat <> people <> that
   <> is <> innocent <> are <> terrorist) M.@@ [S]
>> [[IsTrue]] of grammat type [S] with p=0.5
>> [] of grammat type [S] with p=0.5
```

Other models of meaning

We have been working so far using variations over the category **Rel** of relations. One could argue this is boring and too simplistic: we would like to compare the similarity of two concepts and get more than a simple true or false. But we still have an ace up our sleeve. We implemented everything keeping the compact closed structure abstract, and we can reuse again the same implementation with a different underlying category.

Semirings

Both the category of finite relations **FRel** and the category of finite vector spaces **FVect** can be seen as equivalent to categories of matrices over some semiring: in one case over the booleans $(\mathbb{B}, \vee, \wedge)$, and in the other case over the reals $(\mathbb{R}, +, \cdot)$. We will consider arbitrary semirings in our implementation and play with the different categories we get. In particular, we consider real vector spaces and modules over the *Viterbi semiring*, which has been considered for parsing, for instance, in [Goo99].

Definition 9. A **semiring** (sometimes called a *rig* [nLa18]) is a set R which is a monoid under some multiplication operation and an abelian monoid under some addition operation, in such a way that multiplication distribute over addition.

```
-- Operations of a semiring in Haskell. Checking that they satisfy
-- semiring laws is a task for the programmer.
class Semiring m where
  plus :: m -> m -> m
  mult :: m -> m -> m
  zero :: m
  unit :: m
```

Finite vector spaces over the reals

Finite vector spaces over the real numbers $(\mathbb{R}, +, \cdot)$ are a particular case of the general construction for semirings. We will need to rewrite our universe to account for the structure we want to model. This is just a more sofisticated version of the relations we wrote in Example 2.

```
likes' :: Words (Vectorspace Double)
likes' = Words (fromList
  [ ([Yanek , IsTrue , Dora] , 0.9)
  , ([Dora , IsTrue , Yanek] , 0.8)
  , ([Stepan , IsTrue , Dora] , 0.6)
  , ([Dora , IsTrue , Poetry] , 0.8)
  , ([Dora , IsTrue , Chemistry] , 1)
  , ([Yanek , IsTrue , Poetry] , 1)
  , ([Yanek , IsTrue , Life] , 0.9)
  , ([Dora , IsTrue , Life] , 0.8)
  , ([Stepan , IsTrue , Propaganda] , 0.9)
  , ([Stepan , IsTrue , Life] , 0.1)
  , ([Boris , IsTrue , Life] , 0.3)
  , ([Boris , IsTrue , Propaganda] , 0.6)
  ]) [L N , S , R N]

combat' :: Words (Vectorspace Double)
combat' = Words (fromList
  [ ([Yanek , IsTrue , Duke] , 1)
  , ([Yanek , IsTrue , Skouratov] , 0.7)
  , ([Dora , IsTrue , Duke] , 0.8)
  , ([Dora , IsTrue , Skouratov] , 0.4)
  , ([Stepan , IsTrue , Duke] , 1)
  , ([Stepan , IsTrue , Skouratov] , 0.9)
  , ([Stepan , IsTrue , Nephew] , 0.7)
  , ([Boris , IsTrue , Duke] , 0.9)
  , ([Boris , IsTrue , Nephew] , 0.1)
  , ([Skouratov , IsTrue , Yanek] , 0.9)
  , ([Skouratov , IsTrue , Stepan] , 1)
  ]) [L N , S , R N]

is' :: Words (Vectorspace Double)
is' = Words (fromList
  [ ([Yanek , IsTrue , Revolutionary] , 0.9)
  , ([Yanek , IsTrue , Poet] , 1)
  , ([Dora , IsTrue , Poet] , 0.5)
  , ([Dora , IsTrue , Revolutionary] , 0.7)
  , ([Boris , IsTrue , Revolutionary] , 0.7)
```

```
  , ([Stepan , IsTrue , Terrorist] , 0.95)
  , ([Yanek , IsTrue , Terrorist] , 0.25)
  , ([Boris , IsTrue , Terrorist] , 0.25)
  , ([Stepan , IsTrue , Revolutionary] , 0.8)
  , ([Duke , IsTrue , Tsarist] , 1)
  , ([Skouratov , IsTrue , Tsarist] , 0.9)
  , ([Nephew , IsTrue , Tsarist] , 0.3)
  , ([Nephew , IsTrue , Innocent] , 1)
  , ([Yanek , IsTrue , Innocent] , 0.5)
  ]) [L N , S , R N]

people' :: (Semiring m) => Words (Vectorspace m)
people' = Words (fromList
  [ ([Yanek] , unit)
  , ([Dora] , unit)
  , ([Stepan] , unit)
  , ([Duke] , unit)
  , ([Nephew] , unit)
  , ([Skouratov] , unit)
  , ([Boris] , unit)
  ]) [N]
```

We recover the same sentences from Example 2 and reinterpret them here. Where applicable, we simply repeat the exact same definitions. The scalars are now a bit more informative than our previous true/false booleans.

```
  -- "People that combat Tsarists" lists the revolutionaries and adds
  -- how much they combat each one of the tsarists.
  (people <> who <> combat <> tsarists) M.@@ [N]
  >> [([Yanek],1.63),([Dora],1.16),([Boris],0.9),([Stepan],1.81)]
     of grammar type [N]

  -- "Revolutionaries who enjoy life enjoy propaganda" seems not to
  -- be very true.
  (revolutionaries <> who <> enjoy <> life <> enjoy <> propaganda) M.@@ [S]
  >> [([IsTrue],0.198)] of grammar type [S]

  -- "Revolutionaries who enjoy life enjoy poetry" does much better.
  (revolutionaries <> who <> enjoy <> life <> enjoy <> poetry) M.@@ [S]
  >> [([IsTrue],1.258)] of grammar type [S]

  -- "Revolutionaries who combat people that is innocent are terrorists"
  -- is again true, but it scores not very high because no one really
  -- combats innocent people with much intensity.
  (revolutionaries <> who <> combat <> people <> that <> is
```

```
    <> innocent <> are <> terrorist) M.@@ [S]
>> [([IsTrue],0.549)] of grammar type [S]
```

The Viterbi semiring

One could say that the desired semantics for the addition on the semiring need not to be *addition of real numbers*. For example, when we were computing "*People that combat Tsarists*", we may not want to add how much they combat each one, but just take the maximum as the aggregate. The Viterbi semiring $([0,1], \mathsf{max}, \cdot)$ allows precisely for this and keeps its values into the unit interval.

As our final example, we reinterpret some of the sentences from Example 2 in this third model. The numbers we used for vector spaces can be recycled here, even if they will have a completely different meaning as elements of this new semiring.

```
-- "People that combat Tsarists" lists the revolutionaries but
-- now it takes the maximum instead of adding up values.
(v people <> v who <> v combat <> v tsarists) M.@@ [N]
>> [([Yanek], 1.0),([Dora], 0.8),([Boris], 0.9),([Stepan], 1.0)]
     of grammar type [N]

-- No surprises here. "Revolutionaries who enjoy life enjoy
-- propaganda" gives a slightly lower number because of having
-- substituted addition by maximum.
(v revolutionaries <> v who <> v enjoy <> v life
    <> v enjoy <> v propaganda)
  M.@@ [S]
>> [([IsTrue], 0.198)] of grammar type [S]

-- Same goes for "Revolutionaries who enjoy life enjoy poetry".
(v revolutionaries <> v who <> v enjoy <> v life <> v enjoy <> v poetry)
  M.@@ [S]
>> [([IsTrue], 0.81)] of grammar type [S]

-- A slight difference in the last sentence that can be attributed to
-- the fact that Boris considers the idea of killing innocents if that
-- saves other lifes.
(v revolutionaries <> v who <> v combat <> v people <> v that <> v is
    <> v innocent <> v are <> v terrorist) M.@@ [S]
>> [([IsTrue], 0.532)] of grammar type [S]
```

Conclusions

On the implementation side, dependent types as used in Agda [Nor08] or Idris [Bra13] could be the perfect tool if we want to formally model all the components of the DisCoCat framework. A practical implementation of compact closed categories over dependent types would have been very useful to avoid us repeating some tedious work, and it could be of great utility to the whole applied category theory research community. We are unaware what is the state of the art on this area and unable to tell whether it could help here.

Once the implementation is working, playing with changes in the underlying category is satisfactory. We could have also considered conceptual spaces, or some other of the models proposed in [CGL+18]. On the ambiguity side, we could have chosen to use **density matrices** instead of going with $\mathbf{Rel}_D$, but it was much easier to first think on the simplest model possible. We also got to implement $\mathbf{FVect}_D$ (almost for free thanks to the abstraction layer), but we decided against describing another model of grammatical ambiguity on this project: the main ideas have been discussed in $\mathbf{Rel}_D$ and they would not change much. If we want to extend this work a good idea would be to simply use density matrices for grammatical ambiguity.

Another idea would be to translate the same thing we do for grammatical ambiguity to other monads apart from the finite distribution monad D we used. For instance, it makes sense to consider the free monoid monad (called `List` in Haskell) and let the concatenation of two words return an (unweighted) list of possible grammatical types and meanings; we expect that this would be something like a non-deterministic parsing.

I have enjoyed a lot experimenting with multiple models and the implementation, specially when dealing with relative pronouns. Coming up with some proposal is difficult and I am not very convinced of how my proposed concatenation operation works for multiwords (it is not even associative!). On the other hand, an apology to Camus' fans should be made here; mathematical reality fiercely destroyed what I thought were good examples (with lots of intricacies, and based on the questions posed on the play); and the final examples described on this text look more like a parody of the original. I guess that is part of the charm of trying to write something like this project.

References

[Ash15] Daniela Ashoush. Categorical models of meaning: Accomodating for lexical ambiguity and entailment. *Department of Computer Science*, Trinity 2015.

[BCG+17] Joe Bolt, Bob Coecke, Fabrizio Genovese, Martha Lewis, Dan Marsden, and Robin Piedeleu. Interacting conceptual spaces I : Grammatical composition of concepts. *CoRR*, abs/1703.08314, 2017.

[Bra13] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[Cam49] Albert Camus. *Les Justes*. 1949.

[CGL+18] Bob Coecke, Fabrizio Genovese, Martha Lewis, Dan Marsden, and Alex Toumi. Generalized relations in linguistics & cognition. *Theoretical Computer Science*, 752:104–115, 2018.

[CLM18] Bob Coecke, Martha Lewis, and Dan Marsden. Internal wiring of cartesian verbs and prepositions. *arXiv preprint arXiv:1811.05770*, 2018.

[Coe19] Bob Coecke. The Mathematics of text structure. 2019.

[CSC10] Bob Coecke, Mehrnoosh Sadrzadeh, and Stephen Clark. Mathematical foundations for a compositional distributional model of meaning. *arXiv preprint arXiv:1003.4394*, 2010.

[CU] Cambridge University Press. Cambridge online dictionary.

[Del] Antonin Delpeuch. Type-driven distributional semantics for prepositional phrase attachment. *MSc. Dissertation.*

[Gär04] Peter Gärdenfors. *Conceptual spaces: The geometry of thought.* MIT press, 2004.

[Goo99] Joshua Goodman. Semiring parsing. *Computational Linguistics*, 25(4):573–605, 1999.

[HHJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: Being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, page 1. Microsoft Research, April 2007.

[KSPC14] Dimitri Kartsaklis, Mehrnoosh Sadrzadeh, Stephen Pulman, and Bob Coecke. Reasoning about meaning in natural language with compact closed categories and frobenius algebras. *CoRR*, abs/1401.5980, 2014.

[Lam97] Joachim Lambek. Type grammar revisited. In *International Conference on Logical Aspects of Computational Linguistics*, pages 1–27. Springer, 1997.

[Mar17] Dan Marsden. Ambiguity and incomplete information in categorical models of language. *arXiv preprint arXiv:1701.00660*, 2017.

[nLa18] nLab authors. HomePage. `http://ncatlab.org/nlab/show/HomePage`, May 2018. Revision 262.

[Nor08] Ulf Norell. Dependently typed programming in agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.

[SCC14] Mehrnoosh Sadrzadeh, Stephen Clark, and Bob Coecke. The Frobenius anatomy of word meanings I: subject and object relative pronouns. *CoRR*, abs/1404.5278, 2014.

Appendix: complete implementation

Some implementation choices differ slightly from the theoretical presentation: (1) we have chosen to have a big space called **Universe** that contains both **Nouns** and **Sentences**, even if we avoid having mixed elements in practice; and (2) we prefer to always use multiwords, even when there is no grammatical ambiguity: this eases the implementation.

We think this code could be written in full generality to allow the user to input their own compact closed categories and data over them to test the models. The limited time we have will not allow us to rewrite such a complete implementation.

MainVector.hs

Examples over real vector spaces.

```haskell
{-# LANGUAGE FlexibleInstances          #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE UndecidableInstances       #-}

module MainVec where

import           Lambek
import qualified Multiwords   as M
import           Universe
import           Vectorspaces
import           Words


-- The real numbers have the obvious semiring structure.
instance Semiring Double where
  plus = (+)
  mult = (*)
  unit = 1
  zero = 0

yanek' :: Words (Vectorspace Double)
yanek' = Words (fromList
  [ ([Yanek] , 1)
  , ([Poet]  , 0.7)
  , ([Revolutionary] , 0.9)
  ]) [N]

dora' :: Words (Vectorspace Double)
dora' = Words (fromList
  [ ([Dora] , 1)
  , ([Revolutionary] , 0.9)
  , ([Poet] , 0.3)
  ]) [N]

likes' :: Words (Vectorspace Double)
likes' = Words (fromList
  [ ([Yanek , IsTrue , Dora] , 0.9)
  , ([Dora , IsTrue , Yanek] , 0.8)
  , ([Stepan , IsTrue , Dora] , 0.6)
  , ([Dora , IsTrue , Poetry] , 0.8)
  , ([Dora , IsTrue , Chemistry] , 1)
  , ([Yanek , IsTrue , Poetry] , 1)
  , ([Yanek , IsTrue , Life] , 0.9)
  , ([Dora , IsTrue , Life] , 0.8)
  , ([Stepan , IsTrue , Propaganda] , 0.9)
  , ([Stepan , IsTrue , Life] , 0.1)
  , ([Boris , IsTrue , Life] , 0.3)
  , ([Boris , IsTrue , Propaganda] , 0.6)
  ]) [L N , S , R N]

combat' :: Words (Vectorspace Double)
combat' = Words (fromList
  [ ([Yanek , IsTrue , Duke] , 1)
  , ([Yanek , IsTrue , Skouratov] , 0.7)
  , ([Dora , IsTrue , Duke] , 0.8)
  , ([Dora , IsTrue , Skouratov] , 0.4)
  , ([Stepan , IsTrue , Duke] , 1)
  , ([Stepan , IsTrue , Skouratov] , 0.9)
  , ([Stepan , IsTrue , Nephew] , 0.7)
  , ([Boris , IsTrue , Duke] , 0.9)
  , ([Boris , IsTrue , Nephew] , 0.1)
  , ([Skouratov , IsTrue , Yanek] , 0.9)
  , ([Skouratov , IsTrue , Stepan] , 1)
  ]) [L N , S , R N]

is' :: Words (Vectorspace Double)
is' = Words (fromList
  [ ([Yanek , IsTrue , Revolutionary] , 0.9)
  , ([Yanek , IsTrue , Poet] , 1)
  , ([Dora , IsTrue , Poet] , 0.5)
  , ([Dora , IsTrue , Revolutionary] , 0.7)
  , ([Boris , IsTrue , Revolutionary] , 0.7)
  , ([Stepan , IsTrue , Terrorist] , 0.95)
  , ([Yanek , IsTrue , Terrorist] , 0.25)
  , ([Boris , IsTrue , Terrorist] , 0.25)
```

```haskell
  , ([Stepan , IsTrue , Revolutionary] , 0.8)
  , ([Duke , IsTrue , Tsarist] , 1)
  , ([Skouratov , IsTrue , Tsarist] , 0.9)
  , ([Nephew , IsTrue , Tsarist] , 0.3)
  , ([Nephew , IsTrue , Innocent] , 1)
  , ([Yanek , IsTrue , Innocent] , 0.5)
  ]) [L N , S , R N]

people' :: Words (Vectorspace m)
people' = Words (fromList
  [ ([Yanek] , 1)
  , ([Dora] , 1)
  , ([Stepan] , 1)
  , ([Duke] , 1)
  , ([Nephew] , 1)
  , ([Skouratov] , 1)
  , ([Boris] , 1)
  ]) [N]


yanek = M.singleton yanek'
dora = M.singleton dora'
likes = M.singleton likes'
enjoy = likes
is = M.singleton is'
people = M.singleton people'
combat = M.singleton combat'

who :: (Semiring m) => M.Multiword (Vectorspace m)
who = M.singleton $ Words
  (fromList [ ([a,a,b,a], unit) | a <- universe , b <- universe])
  [L N , N , R S , N]

basis :: Universe -> M.Multiword (Vectorspace m)
basis t = M.singleton $ Words (fromList [ ([t], unit) ]) [N]

tsarist    = basis Tsarist
life       = basis Life
propaganda = basis Propaganda
poetry     = basis Poetry
innocent   = basis Innocent
terrorist  = basis Terrorist


revolutionary :: (Semiring m) => M.Multiword (Vectorspace m)
revolutionary = M.singleton $ Words (fromList [ ([Revolutionary], unit) ]) [N]

tsarists :: M.Multiword (Vectorspace Double)
tsarists = (people <> who <> is <> tsarist) M.@@ [N]

revolutionaries :: M.Multiword (Vectorspace Double)
revolutionaries = (people <> who <> is <> revolutionary) M.@@ [N]

that = who
are = is


-- Tropical semiring. Not used on the examples.
newtype Tropical = Tropical Double deriving (Eq, Show, Num, Ord)
instance Semiring Tropical where
  plus = min
  mult = (+)
  unit = 0
  zero = Tropical $ read "Infinity"
```

MainViterbi.hs

Examples using the Viterbi semiring. These are translates from the examples using vector spaces.

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

module MainVit where

import qualified Data.Map     as Map
import           Lambek
import           MainVec
import qualified Multiwords   as M
import           Vectorspaces
import           Words

-- Viterbi semiring
newtype Viterbi = Viterbi Double deriving (Eq, Show, Num, Ord)
instance Semiring Viterbi where
  plus = max
  mult = (*)
  unit = 1
  zero = 0

-- Reals -> Viterbi translation
v :: M.Multiword (Vectorspace Double) -> M.Multiword (Vectorspace Viterbi)
v = M.fromList . fmap (\ (x , p) -> (v' x , p) ) . M.toList
  where
    v' :: Words (Vectorspace Double) -> Words (Vectorspace Viterbi)
    v' w = w { meaning = v'' (meaning w) }

    v'' :: Vectorspace Double -> Vectorspace Viterbi
    v'' = fromMap . Map.map Viterbi . toMap
```

Main.hs

Examples over the category of relations.

```haskell
  -- We have been used this file for testing the examples.  It does not
  -- contain any interesting code but just some model of the world and
  -- usage examples.
  module Main where

  import           HasCups
  import           Lambek
  import qualified Multiwords as M
  import           Rel
  import           Universe
  import           Words

  -- Example: Yanek attacks the Duke
  yanek :: Words Rel
  yanek = Words yanekRel [ N ]
    where
      yanekRel :: Rel
      yanekRel = fromList
        [ [ Yanek ]
        , [ Poet ]
        , [ Alive ]
        , [ Revolutionary ]
        , [ Innocent ]
        ]

  attacks :: Words Rel
  attacks = Words (fromList
    [ [ Yanek , IsTrue , Duke ]
    , [ Yanek , IsPlot , Duke ]
    ]) [ L N , S , R N ]

  duke :: Words Rel
  duke = Words dukeRel [ N ]
    where
      dukeRel :: Rel
      dukeRel = fromList
```

```haskell
      [ [ Duke ]
      , [ Alive ]
      , [ Tsarist ]
      ]

example2 :: [Words Rel]
example2 = sentence [yanek , kills , duke] @@@ [ S ]


-- Example: Semicartesian verbs
lnot :: Universe -> Words Rel
lnot adjective = Words
  (fromList $ fmap (\x -> [x , x]) $ filter (/= adjective) universe)
  [L N , N]

rnot :: Universe -> Words Rel
rnot adjective = Words
  (fromList $ fmap (\x -> [x , x]) $ filter (/= adjective) universe)
  [N , R N]

cnst :: Universe -> Words Rel
cnst adjective = Words (fromList [[ IsTrue ]]) [ S ]

kills :: Words Rel
kills =
  head $
  sentence [lnot Innocent, cnst IsTrue, rnot Alive]
  @@@ [L N, N, S, N, R N]

example3 :: [Words Rel]
example3 = sentence [ yanek , kills , duke ] @@@ [N , S , N]


-- Example: Grammatical ambiguity (preparation).
nephew :: Words Rel
nephew = Words nephewRel [ N ]
  where
    nephewRel :: Rel
    nephewRel = fromList
      [ [ Nephew ]
      , [ Alive ]
      , [ Tsarist ]
      , [ Innocent ]
      ]

bomb :: Words Rel
bomb = Words
  (fromList [ [Bomb] ])
  [ N ]

and' :: Words Rel
and' = Words
  (fromList $
    [ [ a , a , b ] | a <- universe , b <- universe ] ++
    [ [ a , b , b ] | a <- universe , b <- universe ])
  [L N , N , R N]

example4a :: [Words Rel]
example4a = sentence [ yanek , attacks , duke , and' , nephew ] @@@ [ S ]

using :: Words Rel
using = Words
  (fromList $
    [ [ IsPlot , IsPlot , Bomb ]
    , [ IsTrue , IsTrue , Bomb ]
    ]
  )
  [L S , S , R N]

-- Example: Grammatical ambiguity (full).
with :: M.Multiword Rel
with = M.fromList $
  [ (using , 0.7)
  , (and' , 0.3)
  ]
```

```
yanek'   = M.singleton yanek
duke'    = M.singleton duke
nephew'  = M.singleton nephew
bomb'    = M.singleton bomb
using'   = M.singleton using
and''    = M.singleton and'
attacks' = M.singleton attacks

example5a :: M.Multiword Rel
example5a = (yanek' <> attacks' <> duke' <> with <> nephew')  M.@@ [S]

example5b :: M.Multiword Rel
example5b = M.sentence [yanek' , attacks' , duke' , with , bomb' ] M.@@ [S]


-- Example: Meaning in dispute

-- Yanek is a revolutionary.
-- Yanek kills the duke.
-- Is Yanek a saviour?

becomes :: Words Rel
becomes = Words
  (fromList $
     [ [ a ,     a , b ] | a <- universe , b <- universe ] ++
     [ [ Alive , b , b ] | b <- universe ])
  [L N , N , R N]

becomes' :: M.Multiword Rel
becomes' = M.singleton becomes

revolutionary :: M.Multiword Rel
revolutionary = M.fromList $
  [ ( revSaviour , 0.5 )
  , ( revTerrorist , 0.5 )
  ]
  where
    revSaviour = Words (fromList [[Revolutionary] , [Saviour]]) [ N ]
    revTerrorist = Words (fromList [[Revolutionary] , [Terrorist]]) [ N ]

kills''' :: M.Multiword Rel
kills''' =
  M.singleton $ Words
  (fromList $
    [ [ Alive , Terrorist , Innocent , Innocent ] ]
    ++
    [ [ a , a , b , b  ]
        | a <- universe
        , a /= Innocent
        , b <- universe
        , b /= Alive
        , a /= Saviour
        ]
    ++
    [  ])
  [L N, N, N, R N]

discarding' :: M.Multiword Rel
discarding' =
  M.singleton $ Words
  (fromList
    [ [ a ] | a <- universe ])
  [ L N ]


he = M.singleton $ Words (fromList [[a,a] | a <- universe]) [L N , N]
saviour = M.singleton $ Words (fromList [[Saviour]]) [N]
terrorist = M.singleton $ Words (fromList [[Terrorist]]) [N]
alive = M.singleton $ Words (fromList [[Alive]]) [N]
is' = M.singleton $ Words (fromList [[a,a] | a <- universe]) [L N , N]
(?) = M.singleton $ Words (fromList [[a,a] | a <- universe]) [L N , L N]

example6 :: M.Multiword Rel
example6 = (sentence1 <> sentence2 <> sentence3) M.@@ []
  where
    sentence1 = (yanek' <> becomes' <> revolutionary) M.@@ [N]
```

```
    sentence2 = (he <> kills''' <> duke' <> discarding') M.@@ [L N , N]
    sentence3 = (is' <> he <> terrorist <> (?)) M.@@ [L N]

example7 :: M.Multiword Rel
example7 = (sentence1 <> sentence2 <> sentence3) M.@@ []
  where
    sentence1 = (yanek' <> becomes' <> revolutionary) M.@@ [N]
    both      = (duke' <> and'' <> nephew') M.@@ [N]
    sentence2 = (he <> kills''' <> both <> discarding') M.@@ [L N , N]
    sentence3 = (is' <> he <> terrorist <> (?)) M.@@ [L N]

-- Example: Revolutionaries who kill people who is innocent
people :: M.Multiword Rel
people = M.singleton $ Words
  (fromList
    [ [Yanek]
    , [Dora]
    , [Stepan]
    , [Duke]
    , [Skouratov]
    , [Boris]
    , [Nephew]
    ]
  )
  [N]

combat :: M.Multiword Rel
combat = M.singleton $ Words
  (fromList $
     [ [r         , IsTrue , Duke]      | r <- revolutionaries] ++
     [ [r         , IsTrue , Skouratov] | r <- revolutionaries] ++
     [ [Skouratov , IsTrue , r]         | r <- revolutionaries] ++
     [ [Stepan    , IsTrue , Nephew] ])
  [L N , S , R N]
    where
      revolutionaries = [Yanek, Dora, Stepan]

enjoy :: M.Multiword Rel
enjoy = M.singleton $ Words
  (fromList
    [ [Yanek  , IsTrue , Poetry]
    , [Yanek  , IsTrue , Life]
    , [Dora   , IsTrue , Poetry]
    , [Dora   , IsTrue , Chemistry]
    , [Dora   , IsTrue , Life]
    , [Stepan , IsTrue , Propaganda]
    , [Boris  , IsTrue , Propaganda]
    , [Yanek , IsTrue , Dora]
    , [Dora , IsTrue , Yanek]
    , [Stepan , IsTrue , Dora]
    ])
  [L N , S , R N]

is_ :: M.Multiword Rel
is_ = M.singleton $ Words
  (fromList
    [ [Yanek , IsTrue , Revolutionary]
    , [Yanek , IsTrue , Poet]
    , [Dora , IsTrue , Revolutionary]
    , [Stepan , IsTrue , Revolutionary]
    , [Stepan , IsTrue , Terrorist]
    , [Boris , IsTrue , Revolutionary]
    , [Duke  , IsTrue , Tsarist]
    , [Skouratov  , IsTrue , Tsarist]
    , [Nephew  , IsTrue , Innocent]
    ])
  [L N , S , R N]

who :: M.Multiword Rel
who = M.singleton $ Words
  (fromList [ [a , a , b , a ] | a <- universe , b <- universe ])
  [ L N , N , R S , N ]

tsarist :: M.Multiword Rel
tsarist = M.singleton $ Words (fromList [[Tsarist]]) [N]
```

```
  tsarists :: M.Multiword Rel
  tsarists = (people <> who <> is_ <> tsarist) M.@@ [N]

  revolutionaries :: M.Multiword Rel
  revolutionaries = (people <> who <> is_ <> revolutionary) M.@@ [N]

  example8 :: M.Multiword Rel
  example8 = (people <> who <> combat <> tsarists) M.@@ [N]

  example9 :: M.Multiword Rel
  example9 = (people <> who <> combat <> people <> who <> combat <> tsarists) M.@@ [N]

  -- Example: Revolutionaries who enjoy life enjoy propaganda
  life :: M.Multiword Rel
  life = M.singleton $ Words (fromList [[Life]]) [N]

  propaganda :: M.Multiword Rel
  propaganda = M.singleton $ Words (fromList [[Propaganda]]) [N]

  innocent :: M.Multiword Rel
  innocent = M.singleton $ Words (fromList [[Innocent]]) [N]

  poetry :: M.Multiword Rel
  poetry = M.singleton $ Words (fromList [[Poetry]]) [N]

  chemistry :: M.Multiword Rel
  chemistry = M.singleton $ Words (fromList [[Chemistry]]) [N]

  example10 :: M.Multiword Rel
  example10 = (revolutionaries <> who <> enjoy <> life <> enjoy <> propaganda) M.@@ [S]

  example11 :: M.Multiword Rel
  example11 = (yanek' <> likes <> revolutionaries <> who <> enjoy <> poetry <> or <> chemistry) M.@@ [S]
    where
      likes = enjoy
      or = and''

  -- Main function
  main :: IO ()
  main = return ()
```

Vectorspaces.hs

Cups and identities on the category of vector spaces.

```
module Vectorspaces where

-- Implementation of cups in the category of matrices over a semiring.

import qualified Data.Map   as Map
import           Data.Maybe
import           Dimension
import           HasCups
import           Universe

import           Data.List

class (Eq m, Ord m) => Semiring m where
  plus :: m -> m -> m
  mult :: m -> m -> m
  zero :: m
  unit :: m

data Vectorspace m = Vector (Map.Map UniverseN m)

instance (Show m) => Show (Vectorspace m) where
  show = show . toMap

fromMap :: Map.Map UniverseN m -> Vectorspace m
fromMap = Vector
```

```haskell
toMap :: Vectorspace m -> Map.Map UniverseN m
toMap (Vector v) = v

toList :: Vectorspace m -> [(UniverseN , m)]
toList = Map.toList . toMap

fromList :: (Semiring m) => [(UniverseN , m)] -> Vectorspace m
fromList = fromMap . removeZerosM . Map.fromList . nubPlus
  where
    nubPlus :: (Semiring m) => [(UniverseN , m)] -> [(UniverseN , m)]
    nubPlus = fmap addTogether . (groupBy (\ x y -> fst x == fst y))
    addTogether :: (Semiring m) => [(UniverseN , m)] -> (UniverseN , m)
    addTogether []           = undefined
    addTogether l@((u , x):t) = (u , foldr plus zero (fmap snd l))

removeZerosM :: (Semiring m) => Map.Map UniverseN m -> Map.Map UniverseN  m
removeZerosM = Map.filter (/= zero)

removeZeros :: (Semiring m) => Vectorspace m -> Vectorspace m
removeZeros = fromMap . removeZerosM . toMap

removePlus :: (Semiring m) => Vectorspace m -> Vectorspace m
removePlus = fromList . toList

normalize :: (Semiring m) => Vectorspace m -> Vectorspace m
normalize = removePlus . removeZeros

instance Dim (Vectorspace m) where
  dim = dimVec

dimVec :: Vectorspace m -> Int
dimVec = dimList . Map.toList . toMap
  where
    dimList []      = 0
    dimList (l : _) = length (fst l)

vecCup :: (Semiring m) => Int -> Vectorspace m -> Vectorspace m -> Vectorspace m
vecCup n r s = normalize . fromList . catMaybes . fmap (agrees n) $ do
  (a , x) <- toList r
  (b , y) <- toList s
  return ((a,b) , mult x y)

vecUnit :: (Semiring m) => Vectorspace m
vecUnit = fromList [([], unit)]

agrees :: (Semiring m) => Int -> ((UniverseN , UniverseN) , m) -> Maybe (UniverseN , m)
agrees n ((x , y) , m) =
  if take n (reverse x) == take n y
    then Just $ (reverse (drop n (reverse x)) ++ drop n y , m)
    else Nothing

instance (Semiring m) => HasCups (Vectorspace m) where
  cup   = vecCup
  cunit = vecUnit
```

Rel.hs

Cups and identities on the category of relations.

```haskell
{-# LANGUAGE FlexibleInstances         #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TupleSections             #-}

-- An implementation of the cups and objects of the category of
-- relations.

module Rel
  ( Rel
  , fromList
  , toList
```

```haskell
  , idn
  , relCup
  , agrees
  )
where

import           Data.Maybe
import qualified Data.Set   as S
import           Dimension
import           HasCups
import           Universe

-- A relation hom(1,a) is given by a subset of the universe with
-- elements in a.
data Rel = Rel (S.Set UniverseN)

fromList :: [UniverseN] -> Rel
fromList = Rel . S.fromList

toList :: Rel -> [UniverseN]
toList (Rel u) = S.toList u

instance Show Rel where
  show = show . toList

instance Dim Rel where
  dim = dimRel

dimRel :: Rel -> Int
dimRel = dimList . toList
  where
    dimList []      = 0
    dimList (l : _) = length l

idn :: Int -> Rel
idn n = fromList $ do
  u <- universe
  return $ replicate n u

relCup :: Int -> Rel -> Rel -> Rel
relCup n r s = fromList $ catMaybes $ fmap (agrees n) $ do
  x <- toList r
  y <- toList s
  return (x,y)

relCunit :: Rel
relCunit = fromList [[]]

agrees :: Int -> (UniverseN , UniverseN) -> Maybe UniverseN
agrees n (x , y) =
  if take n (reverse x) == take n y
    then Just $ reverse (drop n (reverse x)) ++ drop n y
    else Nothing

instance HasCups Rel where
  cup = relCup
  cunit = relCunit
```

Lambek.hs

Lambek grammatical types and possible reductions.

```haskell
{-# LANGUAGE FlexibleInstances #-}

-- Lambek grammar types and grammatical reductions for them.

module Lambek
  ( Type (..)
  , Lambek (..)
  , agreeOn
  )
```

```haskell
where

import           Data.List
import           Data.Maybe
import           Dimension
import           HasCups
import           Rel


data Type = N | S | L Type | R Type deriving (Eq, Ord, Show)
type Lambek = [Type]


(>~<) :: Type -> Type -> Bool
a     >~< (L b) = (a == b)
(R a) >~< b     = (a == b)
c     >~< d     = False

agree :: Lambek -> Lambek -> Bool
agree p q = all id $ zipWith (>~<) p q

agreeOn :: Int -> Lambek -> Lambek -> Bool
agreeOn n p q = agree (take n (reverse p)) (take n q)
```

Words.hs

Words as a data structure.

```haskell
{-# LANGUAGE FlexibleInstances #-}

-- Words and how to concatenate them.

module Words where

import           Data.Maybe
import           Dimension
import           HasCups
import           Lambek
import           Rel

data Words m = Words
  { meaning :: m
  , grammar :: Lambek
  }

instance Show m => Show (Words m) where
  show w = show (meaning w) ++ " of grammar type " ++ show (grammar w)

instance Dim (Words Rel) where
  dim = dim . meaning

size :: Words m -> Int
size w = length (grammar w)

maybeCon :: (HasCups m) => Int -> Words m -> Words m -> Maybe (Words m)
maybeCon n u v =
  if agreeOn n (grammar u) (grammar v)
    then Just $ Words
      { meaning = (cup n (meaning u) (meaning v))
      , grammar = reverse (drop n (reverse $ grammar u)) ++ drop n (grammar v)
      }
    else Nothing

tryConcatenate :: (HasCups m) => Int -> Words m -> Words m -> [Words m]
tryConcatenate n a b = catMaybes $ [maybeCon m a b | m <- [0..n]]

concatenate :: (HasCups m) => Words m -> Words m -> [Words m]
concatenate a b = tryConcatenate (min (size a) (size b)) a b
```

```
(@@@) :: [Words m] -> Lambek -> [Words m]
ws @@@ l = filter (\ x -> grammar x == l) ws

(...) :: (HasCups m) => Words m -> [Words m] -> [Words m]
w ... xs = concat $ do
  x <- xs
  return (concatenate w x)

emptyWord :: (HasCups m) => Words m
emptyWord = Words cunit []

sentence :: (HasCups m) => [Words m] -> [Words m]
sentence = foldr (...) [emptyWord]
```

Multiwords.hs

A data structure for multiwords.

```
module Multiwords where

import           Data.List
import           Dimension
import           HasCups
import           Lambek
import           Rel       hiding (fromList, toList)
import           Words

type Probability = Double

-- A multiword is given by a list of different words with different
-- probabilities. Note that these words do not need to have the same
-- grammar types.
data Multiword m = Multiword [(Words m , Probability)]

instance (Show m) => Show (Multiword m) where
  show =
    concat .
    intersperse "\n" .
    fmap (\ (w, p) -> show w ++ " with p=" ++ show p) .
    toList

toList :: Multiword m -> [(Words m , Probability)]
toList (Multiword a) = a

fromList :: [(Words m , Probability)] -> Multiword m
fromList = Multiword

singleton :: Words m -> Multiword m
singleton w = fromList [(w,1.0)]

multiconcat :: (HasCups m) => Multiword m -> Multiword m -> Multiword m
multiconcat x y = fromList $ do
  (w , p) <- toList x
  (v , q) <- toList y
  let concats = concatenate w v
  let newprob = (p * q) / fromIntegral (length concats)
  zip concats (repeat newprob)

infixr 4 multiconcat

multiempty :: (HasCups m) => Multiword m
multiempty = fromList [( emptyWord , 1 )]

instance (HasCups m) => Semigroup (Multiword m) where
  (<>) = multiconcat



instance (HasCups m) => Monoid (Multiword m) where
  mempty = multiempty
```

```
  mappend = multiconcat

sentence :: (HasCups m) => [Multiword m] -> Multiword m
sentence = mconcat

(@@) :: Multiword m -> Lambek -> Multiword m
ws @@ l = fromList $ fmap (\ (x,p) -> (x , p / totalprob)) newlist
  where
    totalprob = sum $ fmap snd newlist
    newlist = filter (\ (x , _) -> grammar x == l) (toList ws)
```

Dimension.hs

Definition of dimension.

```
module Dimension where
class Dim a where
  dim :: a -> Int
```

HasCups.hs

Definition of the cups making a category compact closed.

```
module HasCups where
class HasCups m where
  cup :: Int -> m -> m -> m
  cunit :: m
```

Universe.hs

Our universe of discourse.

```
{-# LANGUAGE FlexibleInstances    #-}
{-# LANGUAGE TypeSynonymInstances #-}

-- A finite universe for the play.
module Universe
  ( Universe (..)
  , universe
  , UniverseN
  , dim
  )
  where

import           Dimension


data Universe
  = Universe

  -- Nouns
  | Yanek
  | Dora
  | Boris
  | Duke
  | Stepan
  | Nephew
  | Skouratov
```

```haskell
  -- Adjectives
  | Poet
  | Revolutionary
  | Terrorist
  | Saviour
  | Innocent
  | Tsarist
  | Alive

  -- Things
  | Life
  | Poetry
  | Chemistry
  | Propaganda

  | Bomb

  -- Sentence meanings
  | IsTrue
  | IsFalse
  | IsRighteous
  | IsWrong
  | IsPlot

  deriving (Eq, Show, Bounded, Enum, Ord)

-- Enumerate all possible values.
universe :: [Universe]
universe = [minBound .. maxBound]

type UniverseN = [Universe]

instance Dim UniverseN where
  dim = length
```