Natürliche Formalisierung der Kategorientheorie

Jonas Benjamin Lippert

Geboren am 1. März 1993 in Münster

April 6, 2021

Bachelorarbeit Mathematik

Betreuer: Prof. Dr. Peter Koepke

Zweitgutachter: Priv.-Doz. Dr. Philipp Lücke

Mathematisches Institut

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Abstract

The subject of this thesis is a formalisation of the Yoneda-Lemma in the controlled, natural language ForTheL. Furthermore, modifications to the proof assistant Naproche are discussed that enable the use of commutative diagrams. This continues current developments that allow the use of `.tex` documents.

Zusammenfassung

Gegenstand der vorliegenden Arbeit ist eine Formalisierung des Yoneda-Lemmas in der kontrollierten, natürlichen Sprache ForTheL. Desweiteren werden Modifikationen am Beweisassistenten Naproche diskutiert, die die Verwendung von kommutativen Diagrammen erlauben. Damit werden aktuelle Entwicklungen fortgeführt, die das Verwenden von `.tex`-Dokumenten ermöglichen.

Contents

1 Introduction

The main reason for formalising mathematical texts is to make them verifiable by proof assistants. Different goals are being pursued in the development of such assistants. For example, the focus of the Lean project is on an efficient implementation that makes it possible to build a large theory of mathematics. A disadvantage here is that a formalisation is very difficult for the reader to understand afterwards. With the Natural Proof Checker Naproche, a working group led by Prof. Koepke is pursuing the approach of enabling the controlled, natural language ForTheL as an input format. ForTheL is an acronym for Formal Theory Language. This is a restricted English that can be parsed, i.e. translated, into a first order formula (FOL). At present, it is possible to formalise individual results in such a way that their appearance differs only insignificantly from the standard literature.

In chapter 2 we want to consider such a formalisation. The formalised theorem is called the Yoneda-Lemma, an early result of category theory. After a short theoretical introduction, the formalisation is discussed in detail. Although we aimed for a presentation of the proof as it is found in [Rie16] we have chosen to follow a single sorted approach as it is presented in [nLa21d]. This means we only have the type of arrows. We expected Naproche to perform better as less structural information has to be processed. Additionally, this approach emphasizes the fact that category theory essentially is a theory of morphisms. Since the Yoneda-Lemma deals with the category of sets, we need to construct an object `SET` that represents the universe of sets but can also be verified as a category in the single-sorted sense. We want to make use of the built-in notions of classes and functions, too. The construction is justified in the corresponding section 2.2.2. The chapter ends with a test for inconsistencies. Hence, for the unlikly event of conflicts arising with the built-in notions, we can be almost sure that the eprover does not use contradictions. Note that our notions are constructable in ETCS. Hence, if contradictions arise, it is in principle possible to replace the built-in notions by new ones.

Chapter 3 is about a diagram parser we added to Naproche. It is able to read out the illustrated equations of diagrams drawn in the tikz-cd environment, as it interprets them as commutative. Like Naproche itself, it is written in Haskell, which is a purely functional programming language. An interesting question is how to integrate such a new module into Naproche. The output is a conjunction of equations, which of course already is part of the standard grammar of ForTheL. Therefore, for our purposes, we have taken the view that commutative diagrams are a metalanguage that we want to pre-parse before we give it to the tokenizer. This is done within the Tokenizer of Naproche. This is a temporary solution, because if the proposed viewpoint is to be investigated further, one has to change the position handling, which is not part of this work. The algorithms are designed by the

author.

Attached to this thesis is a CD-ROM. Any file or document that is mentioned throughout the text can be found on this CD.

We want to end this introductory chapter with a quick installation guide to run a version of ℕaproche within Isabelle jedit with the modifications mentioned above. This works for Linux Mint. For other distributions or when problems may occur, please have a look at `naproche-setup.tar.gz`.

1. Stack: `wget -qO- https://get.haskellstack.org/ | sh`
2. Git: `sudo apt-get install git`
3. Mercurial: `sudo apt-get install mercurial`
4. Build-essential: `sudo apt-get install build-essential`
5. Naproche:
   `git clone https://github.com/naproche-community/naproche.git`
   Copy `Diagram.hs` and `Token.hs` from the enclosed CD and paste them into `.../naproche/src/SAD/Parser`. This will replace the existing file `Token.hs`.
   ```
   cd naproche
   stack clean
   stack build
   ```
6. Isabelle:
   ```
   cd
   hg clone https://isabelle.sketis.net/repos/isabelle-release isabelle
   cd isabelle
   hg update -C -r Isabelle2021
   ./bin/isabelle components -I
   ./bin/isabelle components -a
   ln -s $(pwd)/bin/isabelle  /.local/bin/isabelle
   ```
7. Update reference to Naproche:
   ```
   cd
   isabelle components -u .../naproche # path to Naproche directory
   ```
8. Build Isabelle-Naproche: `isabelle naproche_build`
9. Run jedit: `isabelle jedit`
10. Open the file `Yoneda.ftl.tex` within the editor.

2 About the formalisation

2.1 Content introduction

In this section we want to give a brief intuition for the Yoneda-Lemma. First we want to recall some basic definitions as they are given in [Rie16].

Definition 2.1. A **category** consists of

- a collection of **objects** $X, Y, Z, ...$
- a collection of **morphisms** $f, g, h, ...$

such that:

- Each morphism has specified **domain** and **codomain** objects.
- Each object has a designated **identity morphism** $1_X : X \to X$.
- For any pair of morphisms f, g with the codomain of f equal to the domain of g, there exists a specified **composite morphism** gf whose domain is equal to the domain f and whose codomain is equal to the codomain of g.

This data is subject to the following two axioms:

- For any $f : X \to Y$, the composites $1_Y f$ and $f 1_X$ are both equal to f.
- For any composable triple of morphisms f, g, h, the composites $h(gf)$ and $(hg)f$ are equal.

Definition 2.2. A **functor** $F : C \to D$, between categories C and D, consists of the following data:

- An object $F(c) \in D$, for each object $c \in C$.
- A morphism $Ff : Fc \to Fc' \in D$, for each morphism $f : c \to c' \in C$, so that the domain and codomain of Ff are, respectively, equal to F applied to the domain or codomain of f.

The assignments are required to satisfy the following two **functoriality axioms**:

- For any composable pair f, g in C, $Fg \cdot Ff = F(g \cdot f)$.
- For each object c in C, $F(1_c) = 1_{Fc}$.

Definition 2.3. If C is **locally small**, i.e., between any pair of objects there is only a set's worth of morphisms, then for any object $c \in C$ we define the **covariant functor represented by** c:

$$\begin{array}{ccc} C & \xrightarrow{C(c,-)} & Set \\ \\ x & \longmapsto & C(c,x) \\ \Big\downarrow{\scriptstyle f} & & \Big\downarrow{\scriptstyle f_*} \\ y & \longmapsto & C(c,y) \end{array}$$

where f_* is the post-composition function: $f_*(g) \coloneqq f \cdot g$.

Definition 2.4. Given categories C and D and functors $F, G : C \rightrightarrows D$, a **natural transformation** $\alpha : F \Rightarrow G$ consists of an arrow $\alpha_c : Fc \to Gc$ in D for each object $c \in C$, the collection of which define the **components** of the natural transformations, so that, for any morphism $f : c \to c'$ in C, the following square of morphisms in D

$$\begin{array}{ccc} Fc & \xrightarrow{\alpha_c} & Gc \\ {\scriptstyle Ff}\Big\downarrow & & \Big\downarrow{\scriptstyle Gf} \\ Fc' & \xrightarrow[\alpha_{c'}]{} & Gc' \end{array}$$

commutes.

We are now able to formulate the Yoneda-Lemma.

Lemma 2.5 (Yoneda)**.** *For any functor $F : C \to Set$, whose domain C is locally small, and any object $c \in C$, there is a bijection*

$$Nat(C(c,-), F) \cong Fc$$

that associates a natural transformation $\alpha : C(c,-) \Rightarrow F$ to the element $\alpha_c(1_c) \in Fc$.

We want to visualize the assertion and sketch the proof by the aid of the following figure.

The association $\Phi(\alpha) \coloneqq \alpha_c(1_c)$ is clearly well defined. Our task is to verify for any element $x \in F(c)$, that $\Psi(x)$, defined componentwise by $\Psi(x)_d(f) \coloneqq F(f)(x)$ for each $d \in C$ and $f : c \to d$, is the inverse of Φ. Any element $f : c \to d$ of $C(c,d)$ is sent to $F(f)$ applied on $\Phi(\alpha) = x$. Hence, by naturality of α, which is exemplified by the frontal green square in figure 1,

$$\Psi(\Phi(\alpha))_d(f) = F(f)(\Phi(\alpha)) = F(f)(\alpha_c(1_c)) = \alpha_d(f_*(1_c)) = \alpha_d(f)$$

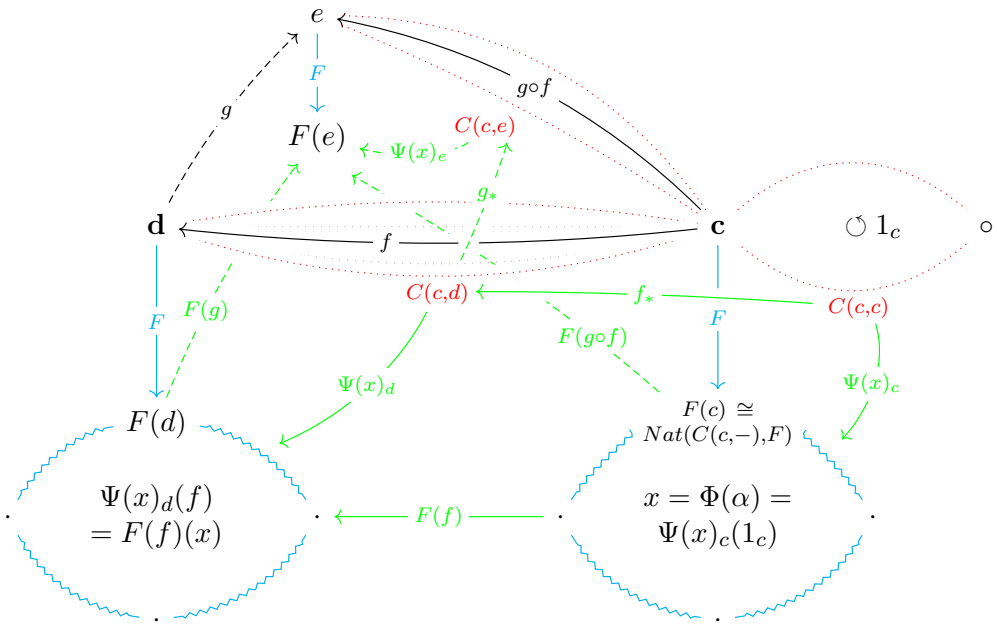


Figure 1: Yoneda-Lemma

as claimed. Further we have

$$\Phi(\Psi(x)) = \Psi(x)_c(1_c) = F(1_c)(x) = 1_{F(c)}(x) = x.$$

To see that $\Psi(x)$ is a natural transformation we have to consider the green square on the left in figure 1. On the right hand side f is sent via $\Psi(x)_d$ to $F(g)(F(f)(x))$. Since the left down direction sends f to $F(g_*(f))(x)$, the claim follows by functoriality of F.

2.2 Documentation

While our aim is the reconstruction of the proof presented by [Rie16], we decided to follow a single-sorted approach. This seems to be more suitable for Naproche to be able to deal with larger expressions towards the end of the formalisation. Therefore the basic definitions are following the text on [nLa21d] for categories and functors. Natural transformations are based on the definition given in [FS90, pg.13]. Since the correspondence is intuitivly obvious and the proof of the equivalence is rather technical, the proof can be found in appendix A.

The onthological check is sometimes a delicate undertaking for Naproche. Especially when the complexity of objects increases, it becomes necessary to repeat earlier results. A good example is the use of the extensionality axiom for functions in the proof of the Yoneda-Lemma. Every variable on which the notion of a natural transformation depends, namely the two categories and two functors, has to be part of a component declaration. Hence, by nesting these objects, the check task only passes after repeating a lot of "type confirmation" statements. Dealing with these nested objects to then use e.g. the axiom above only works when every premise of the axiom is repeated just before the assertion. Although we want to present the complete formalisation, of course we will skip such repetitions. Another type of axioms which we have commented out is the handling of sets as they are currently implemented in Naproche. E.g., to talk about elementhood, every element has to be flagged as `setsized`. We modified the tokenizer such that it ignores `%`. Hence Naproche can still verify the LaTeX document `Yoneda.ftl.tex` while the pdf version does not include these repetitions.

2.2.1 Categories

First we extend our language by some basic notions. Note that the elementhood relation denoted by ϵ is a new one, disinct from the built-in relation symbol $\in$ for sets.

Signature 1. An arrow is a notion.

Signature 2. A collection of arrows is a notion.

Let f, g, h denote arrows. Let C, D denote collection of arrows.

Signature 3. $f \ \epsilon \ C$ is an atom.

Axiom 4. $C = D \iff (f \ \epsilon \ C \iff f \ \epsilon \ D)$.

Signature 5. $s[f]$ is an arrow.

Signature 6. $t[f]$ is an arrow.

Signature 7. $g \circ_c f$ is an arrow.

According to the definition given on [nLa21d], we define a category to be a collection of arrows with the appropriated axioms.

Definition 8. A category is a collection of arrows C such that
(for every arrow f such that $f \in C$ we have
$s[f] \in C$ and $t[f] \in C$ and $t[s[f]] = s[f]$ and $s[t[f]] = t[f]$ and

$$\begin{array}{ccc} s[f] & \xrightarrow{s[f]} & s[f] \\ & \searrow^{f} & \downarrow{\scriptstyle f} \\ & & t[f] \end{array}$$

and

$$\begin{array}{ccc} s[f] & \xrightarrow{f} & t[f] \\ & \searrow^{f} & \downarrow{\scriptstyle t[f]} \quad) \\ & & t[f] \end{array}$$

and (for each arrow f, g such that $f, g \in C$ we have
$(t[f] = s[g] \implies$ (there is an arrow h such that $h \in C$ and

$$\begin{array}{ccc} s[f] & \xrightarrow{f} & s[f] \\ & \searrow^{h} & \downarrow{\scriptstyle g} \\ & & t[g] \end{array}$$

and for every arrow k such that $k \in C$ and $g \circ_c f = k$ we have $h = k)))$
and for all arrows f, g, h such that $f, g, h \in C$ and $t[f] = s[g]$ and $t[g] = s[h]$

$$\begin{array}{ccc} s[f] & \xrightarrow{f} & t[f] \\ {\scriptstyle (g \circ_c f)}\downarrow & & \downarrow{\scriptstyle (h \circ_c g)} \\ t[g] & \xrightarrow{h} & t[h] \end{array} .$$

2.2.2 Construction of SET

To be able to reason about the category of sets, we want to construct it from the notions of sets and functions predefined in Naproche. Since sets in Naproche behave like classes, we have to avoid Russels paradox by defining an `sset` to be a `set` which is contained within another `set`. This is inspired by Kelly-Morse set theory. Functions come along with a predefined

domain `Dom`. We introduce some basic concepts and identify them with the corresponding categorical notions according to our interpretation.

The shortcut at the beginning defines `\innn`, which is on LaTeX level identified with our original relation ϵ.

Let $f \ \epsilon \ C$ stand for (f is an arrow such that $f \ \epsilon \ C$).

Definition 9. An sset is a set x such that x is an element of some set.

Let f, g, h denote functions.

Signature 10. $Cod(f)$ is a notion.

Axiom 11. $Cod(f)$ is a set.

Axiom 12. Let $x \in Dom(f)$. $f(x) \in Cod(f)$.

Axiom 13. (Ext) Let f, g be functions and $Dom(f) = Dom(g)$ and $Cod(f) = Cod(g)$. Let $f(x) = g(x)$ for every element x of $Dom(f)$. $f = g$.

Definition 14. Let $Cod(f) = Dom(g)$.
$g \circ f$ is the function h such that $Dom(h) = Dom(f)$ and $Cod(h) = Cod(g)$ and $h(x) = g(f(x))$ for every element x of $Dom(f)$.

Lemma 15. Let $Cod(f) = Dom(g)$ and $Cod(g) = Dom(h)$.
$h \circ (g \circ f) = (h \circ g) \circ f$.

Axiom 16. Every function is an arrow.

Axiom 17. $s[f]$ is a function such that
$Dom(s[f]) = Dom(f) = Cod(s[f])$.

Axiom 18. $s[f](y) = y$ for every element y of $Dom(f)$.

Axiom 19. $t[f]$ is a function such that
$Dom(t[f]) = Cod(f) = Cod(t[f])$.

Axiom 20. $t[f](y) = y$ for every element y of $Cod(f)$.

Definition 21. $SET = \{$function $f \mid Dom(f)$ is an sset and $Cod(f)$ is an sset $\}$.

Lemma 22. Let $f, g \in SET$ and $Cod(f) = Dom(g)$. $g \circ f \in SET$.

Axiom 23. SET is a collection of arrows.

Axiom 24. Let f be an arrow. $f \ \epsilon \ SET \iff f \in SET$.

Axiom 25. Let $f \ \epsilon \ SET$. $s[f] = Dom(f)$ and $t[f] = Cod(f)$.

Axiom 26. Let $f, g \in SET$ and $Cod(f) = Dom(g)$. $g \circ f = g \circ_c f$.

Justification of SET

Axiom 25 requires special attention. We want to discuss why `SET` corresponds to ordinary set theory. We have functions with domain and codomain that behave like identities. These functions behave like it is required for them to correspond to the single-sorted definition of a category. Hence it is in principle possible to extend the theory of categories (based on these notions) to an Elementary Theory of the Category of Sets (ETCS) as it is discussed in [nLa21a]. Note that, for the moment, this is possible independently from the fact that the domain and codomain have some extra structure (that of an sset). ETCS can be augmented with additional axioms to make it equivalent to full SEARC [nLa21c]. SEARC (Sets, Elements And Relations plus Choice) is another structural theory of sets. Furthermore, SEARC is equivalent to ZFC [nLa21c]. Therefore it is possible to construct the notions of sets and gloabl elementhood within ETCS. We want to sketch the construction of `sset` and $\in$. As suggested in [nLa21c], we want an `sset` to be an equivalence class of well-founded extensional accessible graphs, as described in [nLa21b]:

Definition 2.6. A **graph** is a class G of **nodes** equipped with a binary relation $\to$ on the nodes. A node i is called a **child** of a node j if $i \to j$.

The idea is that nodes $x = s[x]$ represent `ssets`, and $i \to j$ corresponds to $i \in j$.

Definition 2.7. A graph is **well-founded** if it has no infinite paths.

Definition 2.8. A graph is **pointed** if it is equipped with a specified node T called the **root**. A pointed graph is **accessible** if for every node x, there exists a path $x = x_0 \to ... \to x_n = T$ to the root. An accessible pointed graph is abbreviated **APG**.

Such a construction is possible since Th(ETCS) includes the existence of a terminal object $1 = s[1]$:

Axiom 2.9.

$$\exists_1(1 = s(1)) \wedge (\forall_f(f = s(f)) \implies (\exists! g : (s(g) = f \wedge t(g) = 1)).$$

Note that the terminal objects in the ordinary category of sets are the singletons. Hence it makes sence to think of the root of an APG to be its representative.

Definition 2.10. Let G be an APG and z its representative. We define the **full subgraph** of G rooted at y to be the subgraph of G consisting of all those nodes which admit some path to y and write G/y. If y is a child of the root z, we say that G/y is an **immediate subgraph** of G.

These immediate subgraphs represent the `ssets` that are "elements of" G. Note that this is well-defined because such full subgraphs of APGs are APGs themselfes.

Definition 2.11. A well-founded graph is **extensional** if for any nodes x and y such that for all z, we have that $z \to x$ iff $z \to y$ implies $x = y$.

Given this construction, we can interpret our notion of an `sset` as an abbreviatory notation for these graphs. On the other hand, we can think of `SET` to be an ETCS construction within Kelly-Morse (KM) set theory, since ZFC (and thus ETCS) can be proven to be consistent within KM.

We are now ready to prove that `SET` is a category by performing the obvious equations.

Theorem 27. *SET* is a category.

Proof. Let us show that for every arrow f such that $f \in SET$ we have $s[f] \in SET$ and $t[f] \in SET$ and $t[s[f]] = s[f]$ and $s[t[f]] = t[f]$ and $f \circ_c s[f] = f$ and $t[f] \circ_c f = f$.

Proof. Let f be an arrow such that $f \in SET$.

$s[f] \in SET$ and $t[f] \in SET$.

$$(t[s[f]](y) = s[f](y)$$

for every $y \in Dom(f)$) and

$$s[t[f]](y) = t[f](y)$$

for any $y \in Cod(f)$.

We have

$$((f \circ s[f])(y) = f(s[f](y)) = f(y)$$

and

$$(t[f] \circ f)(y) = t[f](f(y)) = f(y))$$

for every $y \in Dom(f)$. Hence $f \circ s[f] = f$ and $t[f] \circ f = f$. Thus

$$\begin{array}{ccc} s[f] & \xrightarrow{s[f]} & s[f] \\ & \searrow^{f} & \downarrow f \\ & & t[f] \end{array}$$

and

$$\begin{array}{ccc} s[f] & \xrightarrow{f} & t[f] \\ & \searrow^{f} & \downarrow{\scriptstyle t[f]} \\ & & t[f] \end{array} .$$

End.

For each arrow f, g such that $f, g \in SET$ we have

($t[f] = s[g] \implies$ (there is an arrow h such that $h \in SET$ and

$$\begin{array}{ccc} s[f] & \xrightarrow{f} & s[f] \\ & \searrow^{h} & \downarrow{\scriptstyle g} \\ & & t[g] \end{array}$$

and for every arrow k such that $k \in SET$ and $g \circ_c f = k$ we have $h = k$)).

Let us show that for all arrows f, g, h such that $f, g, h \in SET$ and t[f]=s[g] and t[g]=s[h] we have $h \circ_c (g \circ_c f) = (h \circ_c g) \circ_c f$.

Proof. Let f, g, h be arrows such that $f, g, h \in SET$ and t[f]=s[g] and t[g]=s[h]. $h \circ (g \circ f) = (h \circ g) \circ f$. Therefore

$$\begin{array}{ccc} s[f] & \xrightarrow{f} & t[f] \\ {\scriptstyle (g \circ_c f)}\downarrow & & \downarrow{\scriptstyle (h \circ_c g)} \\ t[g] & \xrightarrow{h} & t[h] \end{array} .$$

End.

□

2.2.3 Bijections

At this point we give a definition of a bijection via left and right inverse. It is worth mentioning that we do not claim Q to be the codomain of g. It is not necessary for us to enrich our concept of a codomain in a way, to be able to construct it explicitly.

Signature 28. Let Q, R be sets. A bijection between Q and R is a notion.

Axiom 29. Let Q, R be sets. Let f be a function such that $Dom(f) = Q$ and $Cod(f) = R$. Let g be a function such that $Dom(g) = R$ and $g(y) \in Q$ for any element y of R. Let $f(g(y)) = y$ for all elements y of $Dom(g)$. Let $g(f(x)) = x$ for all elements x of $Dom(f)$. Then f is a bijection between Q and R.

2.2.4 Functors

The definition of a functor just differs from [nLa21d] by the fact that a functor does not need to be a function. A functor application is just an operation on arrows.

Signature 30. A functor is a notion.

Signature 31. Let F be a functor. Let f be an arrow. $F[f]$ is an arrow.

Definition 32. Let C, D be categories. A functor from C to D is a functor F such that (for all arrows f such that $f \epsilon C$ we have $F[f] \epsilon D$ and

$$F[s[f]] = s[F[f]]$$

and

$$F[t[f]] = t[F[f]])$$

and for all arrows f, g such that $f, g \epsilon C$ and $t[f] = s[g]$ we have

$$\begin{array}{ccc} F[s[f]] & \xrightarrow{F[f]} & F[t[f]] \\ & \searrow^{F[g \circ_c f]} & \downarrow{\scriptstyle F[g]} \\ & & F[t[g]] \end{array} .$$

2.2.5 Construction of the Hom Functor

Next we have to construct the Hom Functor, also called the covariant functor represented by c.

After defining the collection of morphisms between two identities, we call a category locally small whenever all these collections are elements of SET. Hence we can use them as functions. We define these functions in the way we want the Hom functor to behave. Note that by uniqueness of arrows we have that $s[c] = c$ if $s[h] = c$.

Let C denote a category.

Signature 33. Let $c, x \in C$. $Hom[C, c, x]$ is a collection of arrows such that $f \in C$ for any arrow f such that $f \in Hom[C, c, x]$.

Axiom 34. Let $c, x \in C$. Let h be an arrow.

$$h \in Hom[C, c, x] \iff (s[h] = c \text{ and } t[h] = x).$$

Definition 35. A locally small category is a category C such that $Hom[C, c, f]$ is an element of SET for all arrows c, f such that $c, f \in C$.

Let C denote a locally small category.

Axiom 36. Let $c, x \in C$. Let h be an arrow.

$$h \in Dom(Hom[C, c, x]) \iff h \in Hom[C, c, x].$$

Axiom 37. Let $c, f \in C$.

$$Dom(Hom[C, c, f]) = Hom[C, c, s[f]]$$

and

$$Cod(Hom[C, c, f]) = Hom[C, c, t[f]]$$

and

$$Hom[C, c, f](h) = f \circ_c h$$

for each arrow h such that $h \in Hom[C, c, s[f]]$.

Axiom 38. Let $c, x \in C$. Any element h of $Dom(Hom[C, c, x])$ is an arrow.

The following lemma indicates that our construction can be used to define a functor, which is done afterwards. The verification again is a straightforward calculation.

Lemma 39. (funct) Let $c, f, g \epsilon C$ and $t[f] = s[g]$.

$$Hom[C, c, g] \circ Hom[C, c, f] = Hom[C, c, g \circ_c f].$$

Proof. $f \circ_c h \in Hom[C, c, s[g]]$ for each arrow h such that $h \in Hom[C, c, s[f]]$.

Proof. Let h be an arrow such that $h \in Hom[C, c, s[f]]$. $f \circ_c h$ is an arrow e such that $s[e] = c$ and $t[e] = t[f]$. End.

(h is an arrow and

$$(Hom[C, c, g] \circ Hom[C, c, f])(h)$$
$$= Hom[C, c, g](Hom[C, c, f](h))$$
$$= Hom[C, c, g](f \circ_c h) = g \circ_c (f \circ_c h)$$
$$= (g \circ_c f) \circ_c h = Hom[C, c, g \circ_c f](h))$$

for each element h of $Hom[C, c, s[f]]$. Therefore the thesis (by Ext). □

Definition 40. Let $c \epsilon C$. $HomF[C, c]$ is a functor such that for each arrow f such that $f \epsilon C$ we have

$$HomF[C, c][f] = Hom[C, c, f].$$

Theorem 41. Let $c \epsilon C$. $HomF[C, c]$ is a functor from C to SET.

Proof. For all arrows f such that $f \epsilon C$ we have

$$HomF[C, c][f] \epsilon SET$$

and

$$HomF[C, c][s[f]] = s[HomF[C, c][f]]$$

and

$$HomF[C, c][t[f]] = t[HomF[C, c][f]].$$

Proof. Let $f \epsilon C$.

$$HomF[C, c][s[f]] = Hom[C, c, s[f]] =$$
$$Dom(Hom[C, c, f]) = s[Hom[C, c, f]]$$
$$= s[HomF[C, c][f]].$$

End.

For all arrows f, g such that $f, g \epsilon C$ and $t[f] = s[g]$ we have

$$HomF[C, c][g \circ_c f] = HomF[C, c][g] \circ_c HomF[C, c][f].$$

Proof. Let $f, g \epsilon C$ and $t[f] = s[g]$.

$$Hom[C, c, g] \circ Hom[C, c, f] = Hom[C, c, g \circ_c f]$$

(by funct). End.

□

2.2.6 Natural Transformations

As with categories and the Hom functor, we need to introduce an initial notion, that of a `transformation`, in order to be able to construct such objects. We can then use these objects to check whether they actually match the definition. This definition is based on [FS90] because we do not need the diagonal components which are easy to access given the definition on [nLa21d]. Hence we only have components for identities.

Each `transformation` comes along with a collection of arrows consisting of its components. Since the components depend on the data from the two functors and the two categories, Naproche requires all of these variables to be part of the designation.

Let C, D denote categories.

Signature 42. A transformation is a notion.

Signature 43. Let F, G be functors from C to D. Let α be a transformation. $T[C, D, F, G, \alpha]$ is a collection of arrows.

Signature 44. Let F, G be functors from C to D. Let α be a transformation. Let $d \in C$. $T[C, D, F, G, \alpha, d]$ is an arrow.

Axiom 45. Let F, G be functors from C to D. Let α be a transformation.

$$T[C, D, F, G, \alpha, d] \in T[C, D, F, G, \alpha]$$

for every arrow d such that $d \in C$.

Axiom 46. Let F, G be functors from C to D. Let α be a transformation. For any arrow f such that $f \in T[C, D, F, G, \alpha]$ there exists an arrow d such that $d \in C$ and $T[C, D, F, G, \alpha, d] = f$.

Definition 47. Let F, G be functors from C to D. A natural transformation from F to G over C and D is a transformation α such that
(for any arrow d such that $d \in C$ and s[d]=d we have

$$T[C, D, F, G, \alpha, d] \in D)$$

and
(for any arrow d such that $d \in C$ we have

$$s[T[C, D, F, G, \alpha, d]] = F[d])$$

and
(for any arrow d such that $d \in C$ we have

$$t[T[C, D, F, G, \alpha, d]] = G[d])$$

and for any arrow f such that $f \in C$ we have

$$\begin{array}{ccc} F[s[f]] & \xrightarrow{T[C,D,F,G,\alpha,s[f]]} & G[s[f]] \\ {\scriptstyle F[f]}\downarrow & & \downarrow{\scriptstyle G[f]} \\ F[t[f]] & \xrightarrow[T[C,D,F,G,\alpha,t[f]]]{} & G[t[f]] \end{array} .$$

Definition 48. Let F, G be functors from C to D.

$$Nat[C, D, F, G] = \{ \text{ transformation } \alpha \mid \alpha \text{ is a natural transformation from } F \text{ to } G \text{ over } C \text{ and } D \ \}.$$

Axiom 49. Let F, G be functors from C to D. Let $\alpha, \beta \in Nat[C, D, F, G]$.

$$\alpha = \beta \iff T[C, D, F, G, \alpha] = T[C, D, F, G, \beta].$$

2.2.7 Yoneda-Lemma

To construct the bijection and its inverse we need the use of axioms. That's why they have to be defined beforehand.

Construction of the bijection

When defining functions one has to consider a peculiarity of ℕaproche. The following approach will lead into the undesired situation that every function is identified with the function Ψ:

```
Signature. Ψ is a function.
```

This problem occurs when dealing with sets, too.

Hence we first extend the language by the name of the function. ℕaproche checks well-definedness by itself.

Let C denote a locally small category.

Signature 50. Ψ is a notion.

Axiom 51. Let F be a functor from C to SET. Let $c \in C$. Ψ is a function and $Dom(\Psi) = F[s[c]]$ and $\Psi(x)$ is a transformation for every element x of $Dom(\Psi)$.

Axiom 52. Let F be a functor from C to SET. Let $c, d \in C$. Let $x \in Dom(\Psi)$. $T[C, SET, HomF[C, c], F, \Psi(x), d]$ is a function and

$$Dom(T[C, SET, HomF[C, c], F, \Psi(x), d]) = Hom[C, c, d]$$

and

$$Cod(T[C, SET, HomF[C, c], F, \Psi(x), d]) = F[d].$$

Axiom 53. **(PsiDef)** Let F be a functor from C to SET. Let $c, d \epsilon C$ and $f \epsilon Hom[C, c, d]$. Let $x \in Dom(\Psi)$.

$$T[C, SET, HomF[C, c], F, \Psi(x), d](f) = F[f](x).$$

Signature 54. Φ is a notion.

Axiom 55. Let F be a functor from C to SET. Let $c \epsilon C$. Φ is a function and

$$Dom(\Phi) = Nat[C, SET, HomF[C, c], F]$$

and

$$Cod(\Phi) = F[s[c]].$$

Axiom 56. **(PhiDef)** Let F be a functor from C to SET. Let $c \epsilon C$. Let $\alpha \in Nat[C, SET, HomF[C, c], F]$.

$$\Phi(\alpha) = T[C, SET, HomF[C, c], F, \alpha, s[c]](s[c]).$$

Result

Our previous work enables us to formulate and proof the Yoneda-Lemma as it is presented in [Rie16]. As mentioned before, the most parts of the proof text are redundant. Since improving the management of already known material is an achievable goal in the short term, it is fair to say that ℕaproche is in principle capable of verifying the text as it is presented here. Due to its textbook-like appearance, no further comments seem to be necessary.

Lemma 57. **(Yoneda)** Let C be a locally small category. Let F be a functor from C to SET. Let $c \epsilon C$ and $s[c] = c$.

Φ is a bijection between $Nat[C, SET, HomF[C, c], F]$ and $F[c]$.

Proof. Let us show that for every element x of $Dom(\Psi)$

$\Psi(x)$ is a natural transformation from $HomF[C, c]$ to F over C and SET.

Proof. Let $x \in Dom(\Psi)$.

Let us show that for any arrow d such that $d \epsilon C$ and s[d]=d we have

$$T[C, SET, HomF[C, c], F, \Psi(x), d] \epsilon SET.$$

Proof. Let $d \epsilon C$ and s[d]=d.

$$Hom[C, c, d] = Dom(T[C, SET, HomF[C, c], F, \Psi(x), d])$$

and

$$F[d] = Cod(T[C, SET, HomF[C, c], F, \Psi(x), d]).$$

End.

Let us show that for any arrow d such that $d \epsilon C$ we have

$$s[T[C, SET, HomF[C, c], F, \Psi(x), d]] = HomF[C, c][d].$$

Proof. Let $d \epsilon C$.

$$s[T[C, SET, HomF[C, c], F, \Psi(x), d]] =$$

$$Dom(T[C, SET, HomF[C, c], F, \Psi(x), d]) =$$

$$Hom[C, c, d] = HomF[C, c][d].$$

End.

Let us show that for any arrow d such that $d \epsilon C$ we have

$$t[T[C, SET, HomF[C, c], F, \Psi(x), d]] = F[d].$$

Proof. Let $d \epsilon C$.

$$t[T[C, SET, HomF[C, c], F, \Psi(x), d]] =$$

$$Cod(T[C, SET, HomF[C, c], F, \Psi(x), d]) = F[d].$$

End.

Let us show that for any arrow g such that $g \epsilon C$ we have

$$\begin{array}{ccc} Hom[C, c, s[g]] & \xrightarrow[T[C,SET,HomF[C,c],F,\Psi(x),s[g]]]{} & F[s[g]] \\ {\scriptstyle HomF[C,c][g]}\downarrow & & \downarrow{\scriptstyle F[g]} \\ Hom[C, c, t[g]] & \xrightarrow[T[C,SET,HomF[C,c],F,\Psi(x),t[g]]]{} & F[t[g]] \end{array} .$$

Proof. Let $g \epsilon C$.

$$(T[C, SET, HomF[C, c], F, \Psi(x), t[g]] \circ HomF[C, c][g])(f)$$

$$= T[C, SET, HomF[C, c], F, \Psi(x), t[g]](Hom[C, c, g](f))$$

$$= T[C, SET, HomF[C, c], F, \Psi(x), t[g]](g \circ_c f)$$

$$= F[g \circ_c f](x) = (F[g] \circ_c F[f])(x)$$

for all arrows f such that $f \in Dom(HomF[C,c][g])$.

$$(F[g] \circ T[C, SET, HomF[C,c], F, \Psi(x), s[g]])(f)$$

$$= F[g](T[C, SET, HomF[C,c], F, \Psi(x), s[g]](f))$$

$$= F[g](F[f](x)) = (F[g] \circ_c F[f])(x)$$

for all arrows f such that $f \in Dom(T[C, SET, HomF[C,c], F, \Psi(x), s[g]])$.

End.

QED.

Let us show that for every element x of $Dom(\Psi)$

$$\Phi(\Psi(x)) = x.$$

Proof. Let $x \in Dom(\Psi)$.

$$\Phi(\Psi(x)) = T[C, SET, HomF[C,c], F, \Psi(x), s[c]](s[c]) =$$

$$F[s[c]](x) = s[F[c]](x) = x.$$

QED.

Let us show that for every element α of $Nat[C, SET, HomF[C,c], F]$

$$\Psi(\Phi(\alpha)) = \alpha.$$

Proof.

Let $\alpha \in Nat[C, SET, HomF[C,c], F]$.

For all arrows d, f such that $d \in C$ and $f \in Hom[C,c,d]$ we have

$$T[C, SET, HomF[C,c], F, \Psi(T[C, SET, HomF[C,c], F, \alpha, c](s[c])), d](f)$$

$$= F[f](T[C, SET, HomF[C,c], F, \alpha, c](s[c]))$$

(by PsiDef).

For all arrows d, f such that $d \in C$ and $f \in Hom[C,c,d]$ we have

$$\begin{array}{ccc} Hom[C,c,c] & \xrightarrow{T[C,SET,HomF[C,c],F,\alpha,c]} & F[c] \\ {\scriptstyle Hom[C,c,f]}\downarrow & & \downarrow{\scriptstyle F[f]} \\ Hom[C,c,d] & \xrightarrow[T[C,SET,HomF[C,c],F,\alpha,d]]{} & F[d] \end{array}.$$

Hence

$$F[f](T[C, SET, HomF[C, c], F, \alpha, c](s[c])) = T[C, SET, HomF[C, c], F, \alpha, d](f)$$

for all arrows d, f such that $d \epsilon C$ and $f \epsilon Hom[C, c, d]$.

Therefore

$$T[C, SET, HomF[C, c], F, \Psi(T[C, SET, HomF[C, c], F, \alpha, c](s[c])), d] = T[C, SET, HomF[C, c], F, \alpha, d]$$

for all arrows d such that $d \epsilon C$.

$$\Psi(\Phi(\alpha)) = \alpha \iff$$

$T[C, SET, HomF[C, c], F, \Psi(\Phi(\alpha)), d] = T[C, SET, HomF[C, c], F, \alpha, d]$

for all arrows d such that $d \epsilon C$. Thus we have the thesis. QED.

□

Naproche is able to check this implementation in about 11 minutes on a ThinkPad P53s with an Intel Core i7-8565U CPU @ 1.80GHz 4 and 16GB of RAM. The following is an excerpt from the document `verif`.

```
[Reasoner] "Yoneda.ftl.tex"
verification successful
[Main] sections 348 - goals 90 - trivial 0 - proved 517 -
    equations 0
[Main] symbols 1939 - checks 1900 - trivial 1650 - proved 250
    - unfolds 4859
[Main] parser 00:04.69 - reasoner 00:00.18 - simplifier
    00:00.00 - prover 10:54.27/00:08.12
[Main] total 10:59.15.
```

Note that we had to increase the standard check and proof time and depth. This is done via

```
[checktime 10]
[timelimit 10]
[depthlimit 10]
[checkdepth 10]
```

at the beginning of the document.

2.3 The possibility of inconsistency

Our theory is non-standard. Therefore, we want to complement the justification of our approach in 2.2.2 with an empirical test using the eprover.

To do this, we first pass all the axioms we have established up to the construction of the Hom functor to the eprover (see `Yoneda_axioms1.ftl.tex` and `dump1.p`). After 10 minutes and allocating 12GB RAM, no contradiction can be derived:

```
./eprover --auto --memory-limit=12288 --soft-cpu-limit=600
 dump1.p
...
# Failure: User resource limit exceeded!
```

The complete eprover output can be found in `eprover_output_dump1`.

Similarly, we test our theory by changing each statement to an axiom (see `Yoneda_axioms2.ftl.tex` and `dump2.p`). Again, with 12GB RAM, no contradiction can be shown within 10 minutes

```
./eprover --auto --memory-limit=12288 --soft-cpu-limit=600
 dump2.p
...
# Failure: User resource limit exceeded!
```

See `eprover_output_dump2` for the complete output.

So we can strongly assume that even if our theory is inconsistent and hence most likely not derivable from e.g. ZFC, no contradiction was used for the verification of the argumentation. This is because the hole verification takes less than 11 minutes and the eprover uses a maximum of about 3GB RAM. In this unlikely case, our reasoning in 2.2.2 suggests that only minor changes on the built-in notions would be necessary to restore relative consistency.

3 Documentation of the diagram parser

In this section, we assume basic knowledge of the Haskell programming language in which the parser is written. It can be read as a short introduction into parsing in Haskell.

The diagram parser made it possible to formalize our basic categorical concepts in a more natural way by using commutative diagrams. It can read diagrams that are written in a `tikz-cd` environment which fulfill the following syntax.

- Arrows have to have the form

  `\ar[<options>]{<direction>}[<options>]{<name>}`.

  It is important that the direction is given in the correct order: left, up, right, down. For instance, to write an arrow which points two columns right and one row down, write `rrd`.

- The name of the nodes are irrelevant since it reads only the arrows.
- Names of nodes and arrows must not include braces as they are key characters for the parser.
- The LaTeX code should not include linebreaks if one is interested in correct source positions. This has to do with our temporary solution to track the position.
- Circles are not allowed.
- Columns are separated by &, rows by \\.

The output is a conjunction of equations, where paths with identical start and end points are identified. Because this type of assertion is already part of the grammar of ForTheL, it would be redundant to parse it down to `FTL Formular`. Instead, we'll consider a `tikz-cd` graph as a metalanguage that we want to pre-parse before tokenization. We'll take a closer look at integration later, but for now we just need to know that it's sufficient for the input and output of our parser to be a `string`.

We do not import advanced modules like parsec or attoparsec, since it is part of this work to build such a parser from scratch.

The parser consists of four parts. First, the input is parsed into a `Diagram` type, which is a list of `Arrow`. An `arrow` contains information about the position of its source and target in the form of a tuple. These positions are used to build all possible paths. Then the function

```
mkEqtClass :: (Eq a) => [a] -> [[a]]
```

converts these diagrams into a list of lists of arrows which have common start and end points. Finally, an appropriate function prints out these classes by connecting class members with `=` and classes with `and`.

3.1 Diagram.hs

The goal is to create the parser `tikzcdP`, which translates a diagram in the manner described above. As we want to integrate it later into the tokenizer of Naproche, it is the only function we export from our module. To be able to combine smaller parsers we need `Control.Applicative`. Note that we do not need to import any modules from Naproche as we are simply translating a `String`.

```
module SAD.Parser.Diagram (
  tikzcdP) where

import Control.Applicative
import Data.List
import Data.List.Split
import Data.Maybe
```

3.1.1 Data

We want to define a diagram as a list of arrows. An arrow is a triple consisting of a name, source and target. A position is a tuple of integers as we want to think of our diagram to be a two dimensional object on a grid. We start in the upper left corner on `initpos = (0,0)` and count down each row in the left entry and up each column on the right. Since we want to identify paths with the same source and target, we derive accordingly that `Arrow` is an instance of `Eq`.

```
data Arrow = Arrow
  { name   :: String,
    source :: Position,
    target :: Position
  }
  deriving Show

instance Eq Arrow where
  (==) a b = source a == source b && target a == target b

data Diagram = Diagram
  { arrows :: [Arrow]
  }
  deriving Show

type Position = (Int, Int)
initpos :: Position
initpos = (0,0)
```

3.1.2 Parser

We want a parser to be a function from `String` to `(String, Diagram)`, where the beginning of the input is parsed into a diagram and the rest is kept in the first entry of the tuple for further parsing. Since the beginning of the `String` could be unprocessable for the parser, we have to distinguish between success and failure. Hence we wrap our tuple into the `Maybe` monad. In the case of success we get `Just (String, a)` and otherwise `Nothing`.

```
newtype Parser a = Parser
  { runParser :: String -> Maybe (String, a)
  }
```

So for a general type `a`, a `Parser p` has type `Parser a` if `p` is a function `String -> Maybe (String, a)`. On the other hand, if `p` has type `Parser a` we can extract the function with `runParser p`.
We now want to cunstruct

```
diagramP :: Parser Diagram
diagramP = Diagram <$> many arP
```

by repeating `arP` as often as possible, where `arP` is capable of parsing a single arrow. The `many` method is available for instances of `Alternative`:

```
some v = (:) <$> v <*> many v
many v = some v <|> pure []
```

The details are described below. It applies its argument until it fails and returns the results in a list. Hence `Diagram <$> many arP` is a diagram parser. Since `Alternative` is a monoid on applicative functors, we need to show that `parser a` is an instance of these three classes. In principle, it is only about using the fact that `Maybe` already is such an instance. Thus we are able to manipulate its content inside. E.g., as used above, given a parser `p :: String -> Maybe(String,a)` and a function `f :: a -> b` that we want to apply on the parsing result `x :: a`, we need an appropriate operator `fmap` (identical with `<$>`):

```
instance Functor Parser where
  fmap f (Parser p) =
    Parser $ \input -> do
      (input', x) <- p input
      Just (input', f x)
```

Another operation `<*>` comes with `Applicative`. It allows an application of the parsing result of `Parser p` to that of `Parser q`. Hence we have to

show that such an operation exists.

```
instance Applicative Parser where
  pure x = Parser $ \input -> Just (input, x)
  (Parser p) <*> (Parser q) =
    Parser $ \input -> do
      (input', f) <- p input
      (input'', a) <- q input'
      Just (input'', f a)
```

For illustration, consider `Just (5+) <*> Just 6 = Just 11`. In the definition of `some` we see this operator in action: After the list appending operator `(:)` is lifted on `v`, a list is expected as a second argument which is recursively built by `many v` until `v` is no longer applicable.

Finally, we show that `Parser` is an instance of `Alternative`, which is a straight foreward application of its equivalent for `Maybe`.

```
instance Alternative Parser where
  empty = Parser $ \_ -> Nothing
  (Parser p) <|> (Parser q) =
    Parser $ \input -> p input <|> q input
```

For example, `Nothing <|> Just 5 = Just 5`. We are now able to build parsers for different types and achieved some essential combinators.

3.1.3 arP

The arrow parser waits for braces to get the direction and name of the current arrow. The only problem to be solved is how to carry the position.

```
arP :: Parser Arrow
arP = Parser $ \input -> do
  let pos = pstn input
  (input1, rest)  <- runParser (spanP (/= '{')) input
  let newpos = src rest pos
  (input2, _)    <- runParser (charP '{') input1
  (input3, dr)   <- runParser (spanP (/= '}')) input2
  (input4, _)    <- runParser (spanP (/= '{')) input3
  (input5, _)    <- runParser (charP '{') input4
  (input6, name) <- runParser (spanP (/= '}')) input5
  (input7, _)    <- runParser (charP '}') input6
  case name /= "" of
    True  -> Just (show newpos ++ input7, Arrow name newpos (
    trgt dr newpos))
    _     -> Nothing
```

The idea is to add the current position to the beginning of the unparsed input. This happens in `show newpos ++ input8`. When the next arrow is parsed, we get this position by `pstn input`.

```
pstn :: String -> Position
pstn input@(x:xs) =
  if x == '(' then
    let (a, rest) = span (/= ',') xs in
    let (b, rest') = span (/= ')') (tail rest) in
      (read a, read b)
  else initpos
```

If no position is found, which would be the case for the first arrow, we start at `initpos`. We now have to evaluate the starting point of the current arrow. For this we can use `rest` from

```
(input1, rest)  <- runParser (spanP (/= '{')) input
```

where `spanP` splits the input when { appears for the first time:

```
spanP :: (Char -> Bool) -> Parser String
spanP f = Parser $ \input ->
  let (token,rest) = span f input
  in Just (rest, token)
```

The new position is created by `src`:

```
src :: String -> Position -> Position
src s pos@(a,b) =
  let s1 = splitOn "\\\\" s in
  let cl = length $ filter (=='&') $ last s1 in
    case length s1 of
      1 ->  (a,b+cl)
      _ ->  (a-(length s1)+1,cl)
```

We split on `\\` and get a list `s1` of those strings that were intermitted by `\\`. Since every new line we have to reset the columns, `cl` is the number of & only appearing within the last element of `s1`.

We use `charP` to delete the braces.

```
charP :: Char -> Parser String
charP x = Parser $ \input -> do
  case input of
    y:ys
```

```
      | y == x -> Just (ys, [x])
      | otherwise -> Nothing
    _ -> Nothing
```

After we extracted the name and the direction, `trgt` yields the target position.

```
trgt :: String -> Position -> Position
trgt lurd pos@(a,b) =
  let (lur,d) = span (/= 'd') lurd in
    let (lu,r) = span (/= 'r') lur in
      let (l,u) = span (/= 'u') lu in
        (a + (length u) - (length d), b + (length r) - (length
    l))
```

Finally, if `name /= ""` confirms that an arrow occured, the arrow will be created. Otherwise we get `Nothing` and the `many` operator will stop.

3.1.4 Paths

Once we have our `Diagram`, we need to concatenate the arrows whenever it is well-defined.

```
mkPaths1 :: Arrow -> Arrow -> [Arrow]
mkPaths1 a x
  | source x == target a  = [Arrow ((name x) ++ " \\mcirc " +
   + (name a)) (source a) (target x)]
  | source a == target x  = [Arrow ((name a) ++ " \\mcirc " +
   + (name x)) (source x) (target a)]
  | otherwise             = []

mkPaths2 :: Arrow -> Diagram -> Diagram
mkPaths2 x d = case arrows d of
  [] -> Diagram []
  (y:ys) -> Diagram $ mkPaths1 x y ++ arrows (mkPaths2 x (
   Diagram ys))

mkPaths :: Diagram -> Diagram
mkPaths d = case arrows d of
  [] -> Diagram []
  (x:xs) -> Diagram $ x : arrows (mkPaths (Diagram (xs ++ newA
   )))
    where newA = arrows $ mkPaths2 x $ Diagram xs
```

Lemma 3.1. *The* `mkPaths` *algorithm is correct, that is, every possible path is created, no path will arise twice (efficiency) and the runtime is finite.*

Proof.
Correctness. An arrow x can be considered to be fully integrated into our path system, if it is included by itself and concatenated with any adjacent arrow, since any path that contains x has to contain one of these concatenations. Because `mkPaths` takes the first arrow from the list and prepend it to the final output, the self-includedness is clear. In

```
(x:xs) -> Diagram $ x : arrows (mkPaths (Diagram (xs ++ newA))
```

we call `mkPaths` on the rest of the diagram complemented by `newA`, which is a list of all concatenations of x as required: `mkPaths2` checks for each arrow not equal to x whether concatenation is possible. This is done with the aid of `mkPaths1` which concatenates two arrows via $\circ_c$ as it is needed for our formalisation. Thus, correctness is guaranteed.

Efficiency. The only way for arrows to be added is by `newA`, which extends paths that already exists. A path is characterized by its components and their sequence. Hence concatenation cannot lead to equality between two different paths. Because every arrow in our diagram is presupposed to be unique, we have the thesis.

Finiteness. By efficiency, an infinite runtime presupposes infinite paths. But `tikzcd` graphs are finite and no circles are allowed. □

3.1.5 Equality classes

The path complete diagram has to be partitioned by our predefined equality relation for `Arrow`. Given an arrow x and a list of arrows xs, `mkEqtClass2` compares any arrow of xs with x and creates a list of arrows equal to x. Finally, x is added itself.

```
mkEqtClass1 :: (Eq a) => a -> a -> [a]
mkEqtClass1 x y
  | x == y    = [y]
  | otherwise = []

mkEqtClass2 :: (Eq a) => a -> [a] -> [a]
mkEqtClass2 x xs = case xs of
  [] -> [x]
  (y:ys) -> mkEqtClass2 x ys ++ mkEqtClass1 x y
```

Then `mkEqtClass` creates the first class z by taking the first arrow y from a list of arrows xs and comparing it to rest ys. The recursive call of this function is restricted to our original list minus the generated class. (This behaves well since partitions are disjunct.)

```
mkEqtClasses :: (Eq a) => [a] -> [[a]]
mkEqtClasses xs = case xs of
  [] -> []
  (y:ys) -> z : mkEqtClasses (ys \\ z)
    where z = (mkEqtClass2 y ys)
```

3.1.6 Output

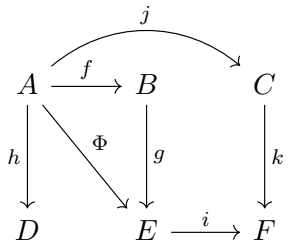
We now apply `mkPaths` and `mkEqtClasses` to our parsing result. The `printEq` function connects class members with `and` and classes with `=`. Note that singleton classes are ignored.

```
tikzcdP :: String -> String
tikzcdP a = maybe "Error: No proper diagram found." (printEq .
    mkEqtClass . arrows . mkPaths . snd) (runParser diagramP a
   )

printEq1 :: [Arrow] -> String
printEq1 l = case l of
  [] -> ""
  [x] -> name x
  (x:xs) -> name x ++ " = " ++ printEq1 xs

printEq :: [[Arrow]] -> String
printEq a = case filter (\l -> length l >= 2) a of
  [] -> ""
  [l] -> printEq1 l -- ++ "."
  (l:ls) -> printEq1 l ++ " and " ++ printEq ls
```

We test our parser using the following diagram:



The parser yields

```
Just ("(-2,1) & F",Diagram {arrows = [Arrow {name = "f",
    source = (0,0), target = (0,1)},Arrow {name = "h", source
     = (0,0), target = (-2,0)},Arrow {name = "\\Phi", source
    = (0,0), target = (-2,1)},Arrow {name = "j", source =
    (0,0), target = (0,2)},Arrow {name = "g", source = (0,1),
     target = (-2,1)},Arrow {name = "k", source = (0,2),
    target = (-2,2)},Arrow {name = "i", source = (-2,1),
    target = (-2,2)}]})
```

The final output is

```
\\Phi = g \\mcirc f and i \\mcirc \\Phi = i \\mcirc g \\mcirc
     f = k \\mcirc j
```

3.2 Integration into ℕaproche

We want the tokenizer to recognize `\begin{tikzcd}` and `\end{tikzcd}` and tokenize the preparsed diagram. After importing `SAD.Parser.Diagram` we have access to `tikzcdP` and use it the following way.

```
-- Pre Parsing of tikzcd
posToken texState pos _ s | start == "\begin{tikzcd}" = toks
      where
          (start, rest) = Text.splitAt 14 s
          (diagram, rest') = Text.breakOn "\end{tikzcd}" rest
          (end, rest'') = Text.splitAt 12 rest'
          diagram' = Text.pack $ tikzcdP (Text.unpack diagram)
          toks = posToken texState (advancePos pos (Text.
    concat [start, Text.replicate ((Text.length diagram) - (
    Text.length diagram')) "d", end])) WhiteSpaceBefore (Text.
    append diagram' rest'')
```

We enter this instance of `posToken` when a tikz-cd evironment was identified. ℕaproche uses `Text` instead of `String`, hence we have to convert the input via `Text.pack` and `Text.unpack` before applying `tikzcdP`. The

converted text is now given to the tokenizer again. The position management is somewhat bulky. We need to subtract the length of the preparsing result whereas source positions are expected to be positive. Therefore, if this way of dealing with meta-language is to be used in the future, it must be accompanied by a revision of `Core.SourcePos`. All diagrams occurring in our formalisation are working fine, but it is of course possible for the result to be longer than the input. Any other way solving this than changing `SourcePos` would be a shift of the problem, hence we leave it at this provisional.

3.3 Additional changes on Naproche

We modified the tokenizer slightly not to treat % as a comment introduction symbol but to use it the way described in the formalisation. Furthermore, we added the possibility to use the `center` environment. It is only about ignoring the associated LaTeXcommands. In this manner, more environments can easily be implemented such as `enumerate` or `align`.

```
-- Ignore texCenter
    posToken texState pos _ s | start == "\begin{center}" =
    toks
      where
        (start, rest) = Text.splitAt 14 s
        toks = posToken texState (advancePos pos start)
    WhiteSpaceBefore rest

    posToken texState pos _ s | start == "\end{center}" =
    toks
      where
        (start, rest) = Text.splitAt 12 s
        toks = posToken texState (advancePos pos start)
    WhiteSpaceBefore rest

-- Ignore %
    posToken texState pos _ s | start == "%" = toks
      where
        (start, rest) = Text.splitAt 1 s
        toks = posToken texState (advancePos pos start)
    WhiteSpaceBefore rest
```

4 Ideas on further developments

We have seen that, at the current stage, Naproche has the capability to verify mathematical texts of moderate length in a legible style as it is able to process LaTeX code rudimentarily. We would like to conclude this work with a list of recommendations for further developments.

- Adding more LaTeX environments such as `enumerate` or `align` can be done by skipping the related commands while tokenizing, like it is presented in chapter 3.3. However, it would certainly be worthwhile to make the automatic numbering usable for Naproche. It would also make sense to allow a quick escape from the Naproche environment, similar to `text` in math mode. This could be a suitable way to integrate explanatory phrases and thus enable the use of more appealing and comprehensible formulations.

- In addition, it would be helpful to receive immediate notification when the eprover yields a contradiction.

- It is important to revise the built-in notion of sets (and functions) so that new objects can be introduced as sets directly through language extensions without being identified with other sets occurring in the text.

- In order to make Naproche available to a wider audience, it is essential that intermediate results are better managed.

- The diagram parser can be extended to parse information from the nodes for two-sorted formalisations. It would also be interesting to be able to use several types of arrows differently, for example to illustrate limits with the help of dashed arrows. Furthermore, one could think about changing the input format, e.g. the output that the app `https://q.uiver.app` yields.

- Generally, one has to consider how meta-languages such as diagrams should be integrated. If the approach proposed in this paper is to be pursued, position tracking must be adapted accordingly.

It is conceivable that Naproche could be used for lectures within the next few years. For example, correcting worksheets could be partially automated. For such everyday operation, it seems essential that proof assistants can work with natural language.

A Equivalence of the single-sorted and the two-sorted theory

For the sake of completeness we want to show that our theory is indeed equivalent to the standard definition. The idea is that sources (and hence targets) correspond to objects.

Theorem A.1. *The single-sorted and the two-sorted definitions of categories, functors and natural transformations are equivalent.*

Proof. In both directions, the composition function and the class of arrows will stay unchanged. Likewise we get uniqueness of arrows as well as associativity and functoriality o arrows immediately.
Now take a model $C = (C_1, s, t, \circ)$ of the single-sorted theory. We define

- the class of objects $D_1 \coloneqq \{Ob_{s[f]} \mid f \in C_1\}$, where $Ob_{(\_)}$ is a type constructor,
- the class of morphisms $D_2 \coloneqq C_1$,
- Dom and Cod to be functions from D_2 to D_1 which maps f to $Ob_{s[f]}$ and $Ob_{t[f]}$ respectively,
- an identity function $Id : D_1 \to D_2$, $Ob_{s[f]} \mapsto s[f]$.

From $s[t[f]] = t[f]$ we see that

$$Id(Ob_{t[f]}) \circ f = t[f] \circ f = f = f \circ s[f] = f \circ Id(Ob_{s[f]}).$$

Hence

$$D = (D_1, D_2, Dom, Cod, Id, \circ)$$

is a model of the two-sorted theory.
Now let $F : C \to C'$ be a functor in the single-sorted sense. We define a functor $H : D \to D'$ that maps each object $Ob_{s[f]} \in D_1$ to $Ob_{s[F(f)]} \in D'_1$ and each function $f \in D_2$ to $F(f) \in D'_2$. This is well-defined since

$$H(Id(Ob_{s[f]})) = H(s[f]) = F(s[f]) = s[F(f)] = Id(Ob_{s[F(f)]})$$

holds for any arrow f.
Finally consider a natural transformation $\alpha : F \Rightarrow G$. We define the two-sorted equivalent $\beta : H \Rightarrow J$ componentwise by $\beta_{Ob_{s[f]}} \coloneqq \alpha_{s[f]}$. Since

$$H(Ob_{s[f]}) = Ob_{s[F(f)]} = Ob_{s[\alpha_{s[f]}]}$$

and

$$J(Ob_{s[f]}) = Ob_{s[J(f)]} = Ob_{t[\alpha_{s[f]}]}$$

we have that $\beta_{Ob_{s[f]}}$ is an arrow from $H(Ob_{s[f]})$ to $J(Ob_{s[f]})$. Naturality of β follows by naturality of α:

$$\begin{aligned} J(f) \circ \beta_{Ob_{s[f]}} &= G(f) \circ \alpha_{s[f]} \\ &= \alpha_{t[f]} \circ F(f) \\ &= \beta_{Ob_{t[f]}} \circ H(f). \end{aligned}$$

Conversly, take a model $D = (D_1, D_2, Dom, Cod, Id, \circ)$ of the standard definition. Again, composition is left unchanged and we define

- $C_1 := D_2$, the collection of arrows,
- $s[f] \coloneqq Id(Dom(f))$, the source of an arrow $f \in C_1$,
- $t[f] \coloneqq Id(Cod(f))$, the target of an arrow $f \in C_1$.

We need to show that $Id(Dom(Id(x))) = Id(x)$ for any object $x \in D_1$. But $Dom(Id(x)) = x$ because $Id(x) \circ f$ is supposed to be well-defined for any $f \in D_2$ such that $f : x \to y$ for some y. Analogously we get $Id(Cod(Id(x))) = Id(x)$ by $Cod(Id(x)) = x$. Hence the identity axioms hold for any $f \in C_1$:

$$t[s[f]] = Id(Cod(Id(Dom(f)))) = Id(Dom(f)) = s[f],$$

$$s[t[f]] = Id(Dom(Id(Cod(f)))) = Id(Cod(f)) = t[f],$$

$$f \circ s[f] = f \circ Id(Dom(f)) = f,$$

$$t[f] \circ f = Id(Cod(f)) \circ f = f.$$

Take a functor $H : D \to D'$. For $F : C \to C'$, $f \mapsto H(f)$,

$$F(s[f]) = H(Id(Dom(f))) = Id(Dom(H(f))) = s[F(f)] \text{ and}$$

$$F(t[f]) = H(Id(Cod(f))) = Id(Cod(H(f))) = t[F(f)]$$

apply as required.
A natural transformation $\beta : H \Rightarrow J$ yields a natural transformation $\alpha : F \Rightarrow G$, that has components for each source, defined by $\alpha_{s[f]} \coloneqq \beta_{Dom(f)}$. Hence, for any arrow $f \in C_1$,

$$\begin{aligned} G(f) \circ \alpha_{s[f]} &= J(f) \circ \beta_{Dom(f)} \\ &= \beta_{Cod(f)} \circ H(f) \\ &= \alpha_{t[f]} \circ F(f). \end{aligned}$$

□

References

[FS90] Peter J. Freyd and Andre Scedrov. *Categories, allegories*. Vol. 39. North-Holland Mathematical Library. North-Holland Publishing Co., Amsterdam, 1990, pp. xviii+296. ISBN: 0-444-70368-3; 0-444-70367-5.

[nLa21a] nLab authors. *fully formal ETCS*. `http://ncatlab.org/nlab/show/fully%20formal%20ETCS`. Revision 31. Mar. 2021.

[nLa21b] nLab authors. *pure set*. `http://ncatlab.org/nlab/show/pure%20set`. Revision 49. Mar. 2021.

[nLa21c] nLab authors. *SEAR*. `http://ncatlab.org/nlab/show/SEAR`. Revision 54. Mar. 2021.

[nLa21d] nLab authors. *single-sorted definition of a category*. `http://ncatlab.org/nlab/show/single-sorted%20definition%20of%20a%20category`. Revision 8. Mar. 2021.

[Rie16] Emily Riehl. *Category theory in context*. Mineola, New York: Dover Publications, 2016.