A new type of automated prover based on category theory

Zijian Wang

Northern Eastern Yucai School, Shenyang, Liaoning, P.R.China

Abstract

In this article I present a new type of automated statement proof algorithm based on new data structures, i.e. brackets and map graphs and new algorithms. The brackets provide an elegant low-knowledge representation of mathematical concepts. The map graphs offer an efficient machine-learning method which let the computer learn knowledge while proving. Additionally, the new finding is totally built on the category theory. Furthermore, a prototype of the program is given.

Introduction

During the long period of development of the research on automated theorem provers, there have already existed a great number of research and implementation on proofs focusing on a specific mathematical subject, such as algebra equations (for example, Wolfram Mathematica's FindEquationProof function [1]) or geometrical theorems (for example, the Geometry Expert or GEX [2] of Key Laboratory of Mathematics Mechanization, Chinese Academy of Sciences). Some relatively new works in this aspect includes Microsoft's Lean prover and Z3 algorithm.

However, if we investigate deeper into these algorithms, we will find the fact that all of these provers stand on the basis of logic theories. Both completely automated theorem provers and semi-automated computer auxiliary statement provers like Coq and Isabelle, are designed to follow certain logic rules and generate results as logic formulas. Nevertheless, although we have to admit that such do is an effective method, this method also sets an insurmountable limit on the execution that the program could not think comprehensively such as solving an algebra problem in a geometrical way.

When we humans think of mathematical problems and perform proofs, we do not follow purely logical ways like complex stuff like Cooper's algorithm. Instead, we just perform tries to seek breakthrough points using his knowledge in a diversified thought. Why cannot programs do things like that? Why can not they perform some sort of knowledge transferring? Carry this question and in experimental and curious attitude, I conducted the research on this topic and have found a utility to do the task: the category theory.

Mainly developed in the twentieth century by the contribution of MacLane and Eilenberg with the purpose to investigate algebra topology, the category theory is a fast-evolving aspect of modern mathematics. In just a few decades, the category theory has become the standard and formal language of homology algebra and algebraic geometry and has gained a lot of meaningful achievements, for instance the Yoneda's Lemma and Braid categories, which serves as a neat explanation for the Yang-Baxter Equation (YBE). Moreover, the category theory's main idea, which is to put mathematical structures into categories satisfies the worldview of the Bourbaki school.

In this paper I will present a new approach of automated proving using knowledge of the category, as well as its theoretical foundation, and will provide a new algorithm which can prove mathematical statements comprehensively. Additionally, an executable program and its implementation process will be describe later.

Structure of the paper.
The paper mainly consists of seven sections. The Introduction section demonstrates the background of the research and fundamental opinions around it. The Preliminaries section, made up of two subsections, claims to minimized pre-knowledge required to understand the paper and the symbols I select to use. The third section defines a set of mathematical structures that will be used by the algorithm. The forth section explains the detailed workflow procedures of the algorithm. The next section claims the conclusions of the research and the to-do improvements in the future. Finally the Acknowledgments section expresses my gratitude to the help and the References section claims the referential resources used while conducting the project.f

Preliminaries

Notations and Conventions

To begin with, we use notation $\mathcal{C}$ to demonstrate a category and use script$\mathcal{F}$ to present a functor. $\mathcal{U}$ stands for the Grothendieck universe selected and $\mathcal{F}or$ stands for the forgetful functor. A map is noted in one of the two following forms:

$$f : A \to B$$

$$x \mapsto y$$

where A and B are sets and$x$and y are elements in A and B.

Furthermore, we use $s(f)$ for the source object of morphism f and use $t(f)$ for the target object of morphism$f$. The symbol $\mathrm{Ob}(\mathcal{C})$ stands for the object collection for the category $\mathcal{C}$. We annotate the morphism set for a category with notation Mor and annotate the $\mathrm{Hom} - \mathrm{set}$ for a category $\mathcal{C}$ between elements a and b with notation $\mathrm{Hom}_{\mathcal{C}}(a, b)$.

Basic logical operators like $\forall$, $\exists$ and $\neg$ are used while basic set operators such as $\in$, $\subset$ and $\cup$ are used too. In addition, we use notation $|S|$ for the size for set S.

For each ordered pair $t = (x, y)$, we use t_ℓ for x and t_r for y.

To avoid misunderstandings, **the Zermelo-Fraenkel set theory** or ZFC as short is utilized to be the set system, with Grothendieck universe concepts added. [3]

Pre-Knowledge

Readers should have basic understandings of the category theory [3] and axiomatic set theory. At least, they should know the basic concepts as well as these conclusions and theorems:

Axiom. The Selection Axiom.

Let X be a set and each of X's elements not empty, then there exists function $g : X \to \cup X$ making $\forall x \in X, g(x) \in x$, naming $g(x)$ as the selection function.

The selection axiom is the ninth axiom in the Zermelo-Fraenkel set system and it is equivalent to the theorem below.

Theorem. The Zermelo's Well-Ordering Theorem. [3]

Every set S can be well-ordering as long as it has a choice function.

Proof: First, we select element $\mho \notin S$. As the ZFC system's selection axiom enables to select element $g(S')$ from each subset S' of S. Let $a_0{:=}g(S)$. Using transfinite recursive theorem to each ordinal α we define:

$$a_\alpha{:=}\begin{cases} g\left(S\backslash\left\{a_\beta|\,\beta<\alpha\right\}\right), & S\neq\left\{a_\beta|\,\beta<\alpha\right\} \\ \mho, & S=\left\{a_\beta|\,\beta<\alpha\right\}\end{cases}$$

Then, since S is a set, a minimized θ can be chosen making $a_\theta=\mho, S=\left\{a_\beta|\,\beta<\theta\right\}$ and significantly there exists a bijection between S and $\theta=\{\beta|\beta<\theta\}$. Following the definition, there obviously exists a well-ordering on S.

■

Some preliminary knowledge also includes the Ebbinghaus Forgetting Curve equation [4] used in this paper later. This approximate function is defined below:

$$B=\frac{100k}{(\ln t)^c+k}$$

Where B is percent of memorization while$t$is time and $c=1.25, k=1.84$. This curve is used by the algorithm for self-optimization purposes.

So as to avoid utilization of impure functions, a pseudo random generator is used. We select the linear congruential generators [5], whose recursive equation is,

$$X_{n+1}=((aX_n+c) \bmod M)$$

Following the ANSI C implementation, we determine the parameters used by the recursive to the followings:

$$M=2^{32}, a=1103515245, c=12345$$

In this paper we use notation r to represent a new random number generated, which is actually a pure function of the last random number and the expression is simplified for representation's explicitness.

The Computerized Representation of Mathematical Structures

Before more specific discussion, we select Grothendieck universe $\mathcal{U}$ [3] to avoid set theory paradoxes. Besides, we define meta category $\mathcal{C}$ where all discussions take place. There are briefly three kind of objects in $\mathcal{C}$: symbols, notations and brackets. All symbols in $\mathcal{C}$ forms sub category $\mathrm{Sym}(\mathcal{C})$ called the **Symbol Subcategory**. All notations in $\mathcal{C}$ forms $\mathrm{Not}(\mathcal{C})$ and all brackets forms $\mathrm{Bra}(\mathcal{C})$, called the **Notation Subcategory** and the **Bracket Subcategory**. That is,

$$\mathcal{C}=\mathrm{Sym}(\mathcal{C})\sqcup\mathrm{Not}(\mathcal{C})\sqcup\mathrm{Bra}(\mathcal{C})$$

The three forms a partition of the big category.

Different from other implementations and research, we describe every major mathematical structure to be used in seeking for proof, from single symbols to contents of proof steps in a single data structure called brackets.

Definition. A **bracket** β is defined as a set of ordered pairs in the form of below:

$$\beta := \{(i, x)\,|i \in \mathrm{On}_{\geq 1}\ , x \in \mathrm{Sym}(\mathcal{C}) \cup \mathrm{Not}(\mathcal{C}) \cup \mathrm{Bra}(\mathcal{C})\}$$

β is valid if and only if every element of β, its left element is unique. We define filtered subsets of β. The **Symbol Subset** of $\beta_{\mathcal{S}} := \{t \in \beta\,|t_r \in \mathrm{Sym}(\mathcal{C})\}$. Following this way, the **Notation Subset** $\beta_{\mathcal{N}}$ and the **Bracket Subset** $\beta_{\mathcal{B}}$ are defined as well. There are two types of brackets: The first type is called a **Symbol Holder** where $\beta = \{(i, A)\}, i \in \mathbb{Z}_{\geq 1}, A \in \mathrm{Sym}(\mathcal{C})$. The second type is called a **Compositor** where

$$\beta = \{(j, \mathcal{N})\} \cup \{(i, x)\,|i \in \mathrm{On}_{\geq 1}\ , x \in \mathrm{Sym}(\mathcal{C}) \cup \mathrm{Bra}(\mathcal{C})\}\,, \forall t \in \beta, j \in \mathrm{On}_{[1, t_\ell]}$$

Here comes an important definition about bracket isomorphism which will be referred inside the third section of this paper.

Definition. Two bracket α and β, are thought to be **isomorphic** (noted as $\alpha \simeq \beta$) if they satisfy the following restrictions:

First, $|\alpha| = |\beta|$ indicating the two brackets have same cardinal numbers. Second, all right elements sorted in the order defined by the sort of the left pair elements, correspondingly, are either same or symbols with the same type. Otherwise, the two brackets are **not isomorphic**, noted as $\alpha \not\simeq \beta$.

Next we define the mathematical representation for notations. A notation is either a operator (for example, $+$ and $\cdot$) or a data type (for example, number and function). Its precise declaration comes below.

Definition. A **notation** $\mathcal{N}$ is defined as an object in $\mathrm{Not}(\mathcal{C})$ which is a sub category of $\mathcal{C}$. The $\mathrm{Ob}(\mathrm{Not}(\mathcal{C}))$ set can be mapped to $\mathrm{On}_{\geq 0}$ class, indicating the number of parameters that a notation take. In which,

$$p : \mathrm{Ob}(\mathrm{Not}(\mathcal{C})) \to \mathrm{On}_{\geq 0}$$
$$\mathcal{N} \overset{p}{\mapsto} n$$

In that case if $n = 0$ then we say $\mathcal{N}$ is a **type** and if $n > 0$ then $\mathcal{N}$ is an **operator**. For instance, obviously, $p(+) = 2$ and $p(\neg) = 1$. A notation$\mathcal{N}$ is called **finite** if $p(\mathcal{N}) < \omega$, and otherwise $\mathcal{N}$ can have infinite number of symbols as parameters, when the notation is called to be **infinite**.

After declaring what notations are, we then define symbols. Basically, a symbol is an instance of a notation. As notations are like containers, as symbol is like the content in them. The accurate representation of a symbol comes below.

Definition. A **symbol** $\mathcal{S}$ is defined as an object in $\mathrm{Sym}(\mathcal{C})$ which is a sub category of $\mathcal{C}$. There exists a functor from $\mathrm{Sym}(\mathcal{C})$ to $\mathrm{Not}(\mathcal{C})$:

$$\nu : \mathrm{Sym}(\mathcal{C}) \to \mathrm{Not}(\mathcal{C})$$
$$\mathcal{S} \overset{v}{\mapsto} \nu(\mathcal{S})$$

Where $\nu(\mathcal{S})$ is the **corresponding notation** of $\mathcal{S}$ and must satisfy $\forall \mathcal{S} \in \mathrm{Sym}(\mathcal{C}), p(\nu(\mathcal{S})) = 0$, indicating all corresponding notations are types instead of operators. For convenience and precision, the text that meaning creating a symbol $\mathcal{S}$ whose corresponding notation is $\mathcal{N}$can and should be written in the form below.

$$\mathcal{S} \in \mathrm{Ob}(\mathrm{Sym}(\mathcal{C})) \wedge v(\mathcal{S}) == \mathcal{N}$$

This should not be written using $\mathcal{S} = v^{-1}(\mathcal{N})$ because if so, the must of $v^{-1}(\mathcal{N}) == v^{-1}(\mathcal{N})$ disables the possibility to create another instance of $\mathcal{N}$, which is ridiculous.

Two brackets are thought to be **equal** when the sequences of t_r, sorted via the ordering of t_ℓ are the same and this relationship constructs a equivalence morphism between the two in the $\text{Bra}(\mathcal{C})$ category, noted as below:

$$\epsilon : \alpha \equiv \beta$$

A bracket, in order to enhance simplicity, can be wrapped into a symbol. Therefore with these three definitions, how do we actually represent mathematical stuff? One example is how to utilize them to express one of the most basic mathematical concepts, such as a single number 12. To complete this task, we may first define a notation Number and define a symbol $\mathcal{S}_1$ whose corresponding notation is Number. Then, how do we determine the number is 12 instead of other numbers like 15? This can be solved using the ordinal theory we presented before in the Preliminary section.

To be more specific and to be accurate, in this system of mathematics, we first need to construct an axiomatic set theory, for example, the Zermelo-Fraenkel set system or we will always encounter the embarrassing situations in the example above. To do this, first, we define a notation called an item (annotated with $\mathcal{I}$) which serves as the parent of all other notations. The parameter count of $\mathcal{I}$ is zero so that it is a type.

Definition. A notation $\mathcal{N}$ is called being **inherited** from notation $\mathcal{M}$ (or we say $\mathcal{N}$ is **a kind of** $\mathcal{M}$) when there exists such morphism between the two in $\text{Not}(\mathcal{C})$:

$$h \in \text{Mor}(\text{Not}(\mathcal{C})) : \mathcal{M} \to \mathcal{N}$$

It is obviously without any ambiguity that $\forall \mathcal{M}, \mathcal{N} \in \text{Not}(\mathcal{C}), |\text{Hom}_{\mathcal{C}}(\mathcal{M}, \mathcal{N})| \in \{0, 1\}$. Moreover, if $\mathcal{N}$ is a kind of $\mathcal{M}$ then $p(\mathcal{N}) \geq p(\mathcal{M})$. For simplicity, $\mathcal{N}$ is inherited from $\mathcal{M}$ can be annotated as $\mathcal{N} \lhd | \mathcal{M}$.

Note that a symbol can be referenced to a bracket, just like the process done below. Let $\emptyset$ be a kind of $\mathcal{I}$, which stands for the empty set. Then we declare $\mathcal{S}et$ as an infinite notation and significantly has $\mathcal{S}et \lhd | \emptyset$. They we define symbol 0 from bracket $\{(i, \emptyset)\}$ where i is an arbitrary positive integer. Following the methods used to define ordinals, we define symbol 1 from bracket $\{(i, \mathcal{S}et), (j, \emptyset)\}$ where $i < j$. Furthermore, 2 is defined via $\{\{\emptyset\}, \emptyset\}$ to be $\{(i, \mathcal{S}et), (j, 1), (k, \emptyset)\}$ where $i < j < k$. So on and so forth, all ordinals can be defined. Moreover, we define symbols declared via brackets in this way Ordinals, using notation $\mathcal{O}$ (obviously a type) so that next time if we need to use an ordinal, it just takes to instance $n \in \text{Sym}(\mathcal{C}) \wedge v(n) == \mathcal{O}$.

Utilizing the structure expressed with Bracket, Notation and Symbol, mathematical statements can be accurately constructed. The example below serves as a instance for this process.

Example. Using the structure provided by Bracket, Notation and Symbol to construct the example formula below, with meta category $\mathcal{C}_1$.

$$\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}, \text{for} |x| < 1$$

First, we need to observe what notation is needed by the construction. Apparently, we need to at least define these notations: infinity, integer, number, $\sum$, $=$, $\hat{}$, $/$, $-$, abs, $<$ and for, where abs stands for the absolute value function. However, in this neat example, the concept of function is not needed to be defined. Additionally, as the constructed stuff only needs to demonstrate the equation itself instead of the principles behind the equality, there is no necessity for the definition of notation $+$. Note that manifestly integer $\lhd |$number. The table below demonstrates the number of parameters taken by the notations.

$\mathcal{N}$	integer	number	infinity	$\sum$	$=$	$/$	$-$	abs	$<$	for
$p(\mathcal{N})$	0	0	0	3	2	2	2	1	2	2

Table 1: The table for $p(\mathcal{N})$

Significantly there are three types and seven operators. In the expressions, symbols are displayed bold for recognition from ordinals, although this is actually not necessary. Bracket α is the result of construction. Then just like constructing a polish expression, we construct:

$$\mathbf{1} \in \mathrm{Ob}(\mathrm{Sym}(\mathcal{C})) \wedge v(\mathbf{1}) = \text{number}$$
$$\boldsymbol{\infty} \in \mathrm{Ob}(\mathrm{Sym}(\mathcal{C})) \wedge v(\boldsymbol{\infty}) = \text{infinity}$$
$$\boldsymbol{x} \in \mathrm{Ob}(\mathrm{Sym}(\mathcal{C})) \wedge v(\boldsymbol{x}) = \text{number}$$
$$\boldsymbol{n} \in \mathrm{Ob}(\mathrm{Sym}(\mathcal{C})) \wedge v(\boldsymbol{n}) = \text{integer} \lhd |\text{number}$$
$$\alpha = \{(0, \text{for}), (1, \beta), (2, \gamma)\}$$
$$\beta = \{(0, =), (1, \delta), (2, \epsilon)\}$$
$$\gamma = \{(0, <), (1, \zeta), (2, \mathbf{1})\}$$
$$\delta = \{(0, \sum), (1, \eta), (2, \boldsymbol{\infty}), (3, \theta)\}$$
$$\epsilon = \{(0, /), (1, \mathbf{1}), (2, \iota)\}$$
$$\zeta = \{(0, \text{abs}), (1, \boldsymbol{x})\}$$
$$\eta = \{(0, =), (1, \boldsymbol{n}), (2, \mathbf{1})\}$$
$$\theta = \{(0, \hat{}\,), (1, \boldsymbol{x}), (2, \boldsymbol{n})\}$$
$$\iota = \{(0, -), (1, \mathbf{1}), (2, \boldsymbol{x})\}$$

This may look a bit messy but nevertheless we can combine the simple brackets together and form larger brackets in the form shown as below. Declaration of the four symbols are omitted. Bracket α is the result of construction.

$$\alpha = \{(0, \text{for}), (1, \beta), (2, \{(0, <), (1, \{(0, \text{abs}), (1, \boldsymbol{x})\}), (2, \mathbf{1})\})\}$$
$$\beta = \{(0, =), (1, \delta), (2, \{(0, /), (1, \mathbf{1}), (2, \{(0, -), (1, \mathbf{1}), (2, \boldsymbol{x})\})\})\}$$
$$\delta = \{(0, \sum), (1, \{(0, =), (1, \boldsymbol{n}), (2, \mathbf{1})\}), (2, \boldsymbol{\infty}), (3, \{(0, \hat{}\,), (1, \boldsymbol{x}), (2, \boldsymbol{n})\})\}$$

It is still possible to combine all into a large bracket definition for α but in that way it may be even harder for people to understand, although altogether no difference for algorithms and programs. (The example ends here.)

Another issue is about the recognition between majorly two types of brackets: the type of bracket that purely serves as a brick for constructing mathematical expressions and the type of bracket that forms the thing that former mathematics call them statements (for instance, $\{(0, =), (1, x), (2, y)\}$ or the expression for the Pythagorean theorem). In this system for representation, the second sort of bracket is called **micro-statements**, whose precise definition comes below.

Definition. A micro-statement is a defined as an object in $\mathrm{Bra}(\mathcal{C})$ whose notation $\mathcal{N}$ has $p(\mathcal{N}) = 2$ and can be evaluated into a Boolean.

It is essential to pay attention to that a bracket with 2-parameter notation is not necessarily a micro-statement. This conclusion can be illustrated via the example of $\{(0,+),(1,x),(2,y)\}$ whose notation has two symbols as parameters but is not a micro-statement. The second restriction is more important, which is that a micro-statement must have the capacity to be evaluated into a Boolean value, that is, true or false. What is notable, the **evaluation** process is not declared inside the representation system, whose definition comes below. All micro-statement brackets forms the **Micro-Statement** sub category of $\mathrm{Bra}(\mathcal{C})$ and is annotated as $\mathrm{Mic}(\mathcal{C})$.

Briefly, an evaluation can be understood by imaging a map which takes a micro-statement bracket and sends a Boolean value as output. The exact declaration is:

Definition. An evaluation is a morphism from $\mathrm{Mic}(\mathcal{C})$ to a Boolean algebra $\mathcal{B}$ with the form below.

$$eva\ell : \mathrm{Ob}(\mathrm{Mic}(\mathcal{C})) \to \mathcal{B}$$

$$m \overset{eva\ell}{\mapsto} eva\ell(m)$$

The Boolean value $eva\ell(m) \in \{0,1\}$ is called the **evaluation result** of micro-statement. If for bracket m exists a morphism to $\mathcal{B}$ that satisfies the definition of evaluation, then we say that m is **evaluatable**. That is, if it satisfies the first restriction of being a micro-statement as well, it is a micro-statement.

The concept of micro-statements are widely and crucially used in the forth section of this paper in the discussion on how to manipulate the data structures described in this section for derivation later.

The Standardized Principles for Derivation of Statements

From the discussion above in the second section, we know that a mathematical statement can be broken into micro-statements, which are a special sort of brackets.

The discussion below calls the $\mathrm{Mic}(\mathcal{C})$ category the **micro-statement pool**, or **pool** for short. The morphisms inside the micro-statements pool are declared for representation of implication relationships. It is obvious are elementary that,

$$\forall \alpha, \beta \in \mathrm{Ob}(\mathrm{Mic}(\mathcal{C})), \forall \left|\mathrm{Hom}_{\mathrm{Mic}(\mathcal{C})}(\alpha,\beta)\right| = 1$$

Since for each micro-statement α and β, we only need one arrow for derivation $\alpha \Rightarrow \beta$.

The basic insight of the entire derivation process is defined below.

Definition. A **derivation** is a set of ordered pairs of ordinals and categories in the form below.

$$\mathcal{D} := \{(i, \mathcal{M}_i)\,|i \in \mathrm{On}_{\geq 1}\}$$

Where $\mathcal{M}_i$ is a micro-statement pool and for each element in $\mathcal{D}$, its left pair element is a unique ordinal. The $\mathcal{D}$ set is well-ordered following the sort of i. On the beginning, the the initial pool equals to the condition, that is the micro-statements given by the problem. On the other hand, the conclusions of the proof problem, to be proved via some steps form a set $\mathcal{R}$, called the **requirement set** are to be inferred to exist inside the micro-statement pool.

For simplicity of expression, we use notation $\mathcal{M}_i$ to represent the right pair element in $\mathcal{D}$ whose left companion is i.

The detailed explanation of derivation, illustrating how the algorithm works from the ground up is defined below.

Definition. A **step of derivation** is a map existed on the derivation set $\mathcal{D}$, mapping from a pair to another, whose ordinal increases by 1. This map is annotated using script word *step*. The form is shown below:

$$(i, \mathcal{M}_i) \overset{step}{\mapsto} (i+1, \mathcal{M}_{i+1})$$

Inside the map, the left pair element is simply self-increased and the right pair element is processed by a functor (since the item to process is a category) called the **recurse functor** noted as $\mathcal{F}_i$. That is,

$$\mathcal{M}_{i+1} = \mathcal{F}_i \mathcal{M}_i$$

The termination condition for the proof automation process is described below as an evaluatable statement.

Definition. A derivation set is thought to be **successful** when the condition below is satisfied when the condition below is true:

$$\exists n \in [0, \omega), \mathcal{R} \subseteq \mathrm{Ob}\left(\mathcal{M}_n\right)$$

On the other hand, a derivation set is thought to be **impossible** when there does not exist such a $n \in [0, \omega)$ that makes the requirement set a subset of the object set of the targeted micro-statement pool.

It should be noted that the impossibility detection of a derivation set, although is mathematically well-defined, is not actually applicable to being implemented in the algorithm because a computing algorithm's calculation and derivation time is limited, just like Turing's halting problem, cannot be judged. Then we investigate into the execution of the recurse functor, which is significantly the core part that performs the proving process.

Another notable issue about the recurse functors is that in order to reduce stubbornness, the functors are impure is there is no global variable. To solve this problem, we utilize the r random number function utility described in the Preliminary section.

Definition. A **reflection functor** $\mathcal{K}$ is defined as an functor object in a functor category $\mathcal{R}ef$, who maps between two micro-statement pools. Because of the Zermelo's Well-Ordering Theorem we mentioned in the Preliminary section, $\mathrm{Ob}(\mathcal{R}ef)$ can be put into well-order, with ordinal i. All of the reflection functors forms a functor category $\mathcal{Q}$, called the **reflection knowledge base**, or for short the **knowledge base**. The detailed explanation and description, including the utilization of and morphisms inside the knowledge base, will be discussed later in this section of the paper. We annotate each of the functor positioned at location i with $\mathcal{K}_i$.

Each reflection functor corresponds to a **template pair** of micro-statements (σ, τ) which indicates that the effect of the functor is to transform micro-statement σ into τ. The template pair of functor $\mathcal{K}$ is annotated with $T(\mathcal{K})$. That is,

$$\sigma = T(\mathcal{K})_\ell$$
$$\tau = T(\mathcal{K})_r$$

When reflection functor $\mathcal{K}_i$ is applied to a pool $\mathcal{M}_i$, each of the micro-statements in the object set of the pool is proceeded. If a micro-statement is suitable for the reflection, whose detailed explanation will be discussed later, a new micro-statement will be created if there's no duplication. Then a morphism will be built from the old micro-statement to the new item. The process of identifying whether a micro-statement is capable called an **isomorphism inspection** is described as below.

Definition. The process of isomorphism inspection is declared as below. The stuff that the isomorphism inspection process manipulates is a micro-statement $\mu \in \mathcal{M}_i$, a special evaluatable sort of bracket. The definition of bracket being isomorphic can be found in the second section. Explicitly, the inspection process is to construct a new set called the **transformation source**, noted as Ts using the equation below.

$$\text{Ts:=} \{t \in \mathcal{M}_i | t \simeq T(\mathcal{K})_\ell\}$$

What to do next is called a **transformation**. Foremost, creates a set of transformation rules that makes one-one maps the micro-statement to manipulate to the left of the template pair σ. Secondarily, the reverse map is applied to the right of the template pair τ and then the transformation process execution is done. Then we delineate the definition of transformation mapping.

Definition. A **transformation map** is a one-one map defined between the list of symbol and sub-brackets of the two brackets α and β where,

$$\mathcal{T} : \alpha_\mathcal{B} \cup \alpha_\mathcal{S} \to \beta_\mathcal{B} \cup \beta_\mathcal{S}$$
$$\mathcal{S}_1 \overset{\mathcal{T}}{\mapsto} \mathcal{S}_2$$

To begin with, the translation map between the two temporary sets, that is each of the elements of transformation source and the left element of the template pair, are constructed forming an executable rule that makes each non-notation right pair element x in $\mu\epsilon\mathcal{M}_i$, which is a micro-statement to discuss, to have $\mathcal{T}(x) \in \sigma_\mathcal{B} \cup \sigma_\mathcal{S}$. After the construction, we execute a **reverse transformation** from τ to a new micro-statement μ' satisfying that

$$(\mu')_\mathcal{B} \cup (\mu')_\mathcal{S} = \mathcal{T}^{-1} (\mu_\mathcal{B} \cup \mu_\mathcal{S})$$

And then micro-statement μ' is the freshly baked derivation achievement. All of the micro-statements as results of map $\mathcal{T}^{-1}$ forms new set $\mathcal{M}'$. In the end we process the pool $\mathcal{M}$ with this union operation:

$$\mathcal{M}_{i+1} = \mathcal{M}_i \cup \mathcal{M}'$$

The process above explains how the reflection functor $\mathcal{K}$ evolves $\mathcal{M}_i$ into $\mathcal{M}_{i+1}$. Nevertheless, actually the derivation is be done into practice using recursive functor $\mathcal{F}_i$, which select $\mathcal{K}$ as the reflection functor to utilize, following the specification we had like to present below.

The recursive functor $\mathcal{F}_i$ decides the reflection functor to use based on these things: the random seed (in this paper is passed implicitly and the random generator function is annotated with r. However, the functor is for sure pure because actually the seed is passed as a parameter), knowledge base that is the functor category that all reflection functors form, as well as an integer j, indicating last time $\mathcal{K}_j$ is executed. To begin with, we will investigate into the organization of objects in the knowledge base $\mathcal{Q}$.

Definition. The **initial map** is the original state of knowledge base $\mathcal{Q}$ when $\mathcal{Q}$ is created. Initially, after the reflection functors are sorted, while discussions around this take place before, they forms a circular with connection of morphisms. That is, for $N = |\text{Ob}(\mathcal{Q})|$,

$$\mathcal{K}_n \overset{k_n}{\to} \mathcal{K}_{n+1}, n \in \mathbb{Z}_{[0,N-1]}$$
$$\mathcal{K}_N \overset{k_N}{\to} \mathcal{K}_0$$

Graphically, we have:

$$\begin{tikzcd}
 & & \mathcal{K}_0 \arrow[dll, "k_0"] & & \\
\mathcal{K}_1 \arrow[dr, "K_1"] & & & & \mathcal{K}_N \arrow[ull, dashed, "k_N"'] \\
 & \mathcal{K}_2 \arrow[rr, "k_2"] & & \mathcal{K}_3 \arrow[ur, dashed, "K_3"] &
\end{tikzcd}$$

Figure 1: The initial map.

A morphism in this category means that if the reflection functor on its start is executed just now, then the target functor will be executed next time. That is, to be more specific, if a functor in $\mathcal{Q}$ has no morphisms targeting it, then it will be never executed unless it is configured as the beginning functor (location 0) manually. Moreover, if a functor has more than one morphisms starting from it, then we call this functor **extroverted**. Otherwise, if a functor has more than one morphisms targeting it, it is **introverted**. If one has only two morphisms connected, one in and one out, thus it is **ordinary**. The process of determining the next functor of a extroverted functor will be discussed below.

The definition below describes how the recursive functor uses the knowledge base and the further evolution for the data structure of the knowledge base. It ought to be paid attention to that although in the Introduction part that I say this new method of implementing a prover makes the prover low-knowledge and it turns out to have an infrastructure called the knowledge base. This is not conflictual since the knowledge I mean by in the Introduction section means the algorithm is automated, which means it does not require users to give in too much assistance and it does not know exactly what is under manipulation. On the other hand, however, the knowledge word here means by the storage of reflections, which can be rules, axioms, theorems, lemmas and even strategies.

Definition. A recursive functor $\mathcal{F}_i$ is a functor that applies to pool $\mathcal{M}_i$ and evolves it into $\mathcal{M}_{i+1}$. The process of determining which $\mathcal{K}_w$ is **selected** after former reflection functor $\mathcal{K}_j$ generally works on the knowledge base $\mathcal{Q}$, right starting from the initial map. If this is the first time for the recursive functor to execute, then $\mathcal{K}_0$ is selected. If $\mathcal{F}_i$ encounters with an introverted or ordinary functor $\mathcal{K}_j$ then significantly,

$$w = k_j\left(\mathcal{K}_j\right)$$

If $\mathcal{F}_i$ encounters with an extroverted functor, then the random number r is utilized and w turns out to be the following:

$$w = (r \bmod \sum_{u=0}^{N}) \left|\mathrm{Hom}_{\mathcal{Q}}\left(\mathcal{K}_j, \mathcal{K}_q\right)\right|$$

It is easy to understand that for each reflection functor $\mathcal{K}_a$ and $\mathcal{K}_b$, the count of set $\mathrm{Hom}_{\mathcal{Q}}\left(\mathcal{K}_a, \mathcal{K}_b\right)$ indicates the **weight** of $\mathcal{K}_b$ to $\mathcal{K}_a$, controlling the probability of $\mathcal{K}_b$ to be selected when the former functor is $\mathcal{K}_a$. The design of this process is inspired by the lottery scheduling algorithm I learned in Andrew S. Tanenbaum's Operating Systems: Design and Implementation [6] before.

An issue that worth attention is that as the random generator is defined as the following recursion formula: [5]

$$X_{n+1} = ((1103515245X_n + 12345) \bmod 2^{32})$$

Then each r must belongs to region $\mathbb{Z}_{[0,2^{32})}$. That is, obviously, if $\sum_{u=0}^{N} |\mathrm{Hom}_{\mathcal{Q}}(\mathcal{K}_j, \mathcal{K}_q)| \geq 2^{32}$, then there will be some of the functors that will never be referred to, although there do exist morphisms targeting them. While in this paper the problem will not be solved and we assume that the number of the morphisms added up is smaller than the boundary of 2^{32}, which equals to four giga bytes and will not be exceeded in most engineering situations. Obviously although using the portable recursion equation the quality of the random numbers generated may be lower than nowadays' some technical implementations, it can be conveniently expressed in this paper. If the algorithm is implemented into the read world, a randomness collector, which generates random numbers from the real world, for example, the viberation of the computer motherboard, will probably act as a better solution.

It should be noted that if the initial map does not evolve, the proof efficiency will be primitive. Hence, we define a supplement process called **knowledge reinforcement**, with definition coming below.

Definition. The process of knowledge reinforcement is done after a finishing a proof. The first step of this reinforcement is to add numbers of utilization of the corresponding morphism of copies of morphisms into the knowledge base, increasing the weight of connections between the reflection functors. When next time a proof is issued, the modified knowledge base is **loaded** into instance and act as the modified initial situation.

The second step of this sort of reinforcement is called **reduction**, which is based on the Ebbinghaus's forgetful curve's numerical fit function. The exact form of the function expression can be found in the Preliminary section. Specifically, the number of morphisms is decreased following the forgetful equation, changing into the nearest integer number. To be more explicit, the Ebbinghaus-reduced decimal is stored separately from the knowledge base and the exact and accurate representation for this is omitted in the paper.

A situation after some reinforcements may be the one demonstrated in the figure below.

$$\begin{tikzcd}
 & & \mathcal{K}_0 \arrow[dll, "k_0"] \arrow[ddr, "k_N"] & & \\
\mathcal{K}_1 \arrow[rrrr] \arrow[dr, "K_1"] \arrow[drrr] & & & & \mathcal{K}_N \arrow[ull, dashed] \\
 & \mathcal{K}_2 \arrow[rr, "k_2"] & & \mathcal{K}_3 \arrow[ur, dashed, "K_3"] &
\end{tikzcd}$$

Figure 2: An example map after several reinforcements.

Via the process we described in the forth section and the data structures we presented in the third section, a mathematical proof can be conducted over category theory's accurate representation. Then, a proof procedure is generated using the form of reflection history, that is, the historical list of the reflection functors utilized by the proof.

Conclusions and Future Work

In the sections before I have presented an elegant data structure for representing mathematical concepts and a neat process which utilizes category theory concepts and knowledge to conduct proof, which also have self-optimization features.

Currently I have developed a computer program called DefQed that implements the algorithm and it is open-sourced with BSD 3-Clause “New” or “Revised” License on GitHub. The source code of the latest version can be accessed at [7] It should be noted that the version is very early and some aspects of the algorithm I presented in the paper is not implemented.

Future works are the followings below:

- **Continuing the implementation of the corresponding software.** Currently the software I developed is still under immature development and many of the aspects I presented in the paper above, including the self-evolution logic, have not been constructed yet.
- **Improving the algorithm.** The algorithm I presented in the paper is not mature. For instance, if the knowledge base has more than 2^{32} morphisms, then some of the morphisms will never be utilized.
- **Optimizing the performance of the algorithm.** Currently, though the design of the procedures increase generality because of the low-knowledge feature, the performance of derivations may be lower than the current algorithms, which focus on a certain subject inside mathematics.

Acknowledgments

I express my sincerest greatest gratitude to my tutor Prof. Shao Xinhui, the Yingcai Project of P.R.China, which enables senior high school students to conduct science research, my senior high academy Northern Eastern Yucai School (NEYC) of P.R.China and the corresponding university Northern Eastern University (NEU) of P.R.China. I also thank my parents for encouraging me to conduct this research project.

References

[1] S. Wolfram, *Wolfram Language & System Documentation Center.* Wolfram Research Inc., 2019.

[2] X.-S. Gao, “Geoexpert,” 1998.

[3] W.-W. Li, *Daishuxue Fangfa.* High Education Press, 1 ed., 2019.

[4] H. Ebbinghaus, *Memory.* Teachers College, Columbia University, 1913.

[5] K. Entacher, “A collection of selected pseudorandom number generators with linear structures,” 1999.

[6] A. S. Tanenbaum, *Operating Systems: Design and Implementatiion.* Publishing House of Electronics Industry, 3 ed., 2015.

[7] Z. Wang, “The defqed repository,” 2022.