AIMS Mathematics

http://www.aimspress.com/journal/Math

AIMS Mathematics, 8(8): 18278–18294.
DOI: 10.3934/math.2023929
Received: 08 January 2023
Revised: 29 April 2023
Accepted: 19 May 2023
Published: 29 May 2023

Research article

A new type of generic, self-evolving and efficient automated deduction algorithm based on category theory

Zijian Wang[1] **and Xinhui Shao**[2,*]

[1] Northeast Yucai School, No.2, Gaogong Road, Shenyang, Liaoning, China

[2] Department of Mathematics, College of Sciences, Northeastern University, Shenyang, Liaoning, China

* **Correspondence:** Email: shaoxinhui@mail.neu.edu.cn.

Abstract: In this article, a new type of generalized, self-evolving and efficient automated statement proof algorithm based on new data structures, i.e., brackets and map graphs, and new algorithms is presented. The brackets structure provides 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 built completely on category theory. Furthermore, a prototype of the program is presented and examined for performance.

Keywords: automatic prover; theorem prover; algorithm; category theory; automatic deduction; automatic reasoning
Mathematics Subject Classification: 03B35, 68V15, 18-00, 18-04, 68W99

1. Introduction

During the long period of development and 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 algebraic equations (for example, Wolfram Mathematica's Find Equation Proof 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 included Microsoft's Lean prover [3] and the Z3 [4] 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 it is an effective method, this method also sets an insurmountable limit on the program that it can not think comprehensively such as when solving an algebra problem in a geometrical way.

When we humans think of mathematical problems and perform proofs, we do not follow purely logical methods, for example, by executing the Cooper's algorithm. Instead, we attempt to seek breakthrough points using our knowledge and diverse thought. Why can programs not do things like that? Why can not they perform some sort of knowledge transfer? Interested in these questions, we conducted research on this topic and have found a utility to perform the task: Category theory.

Mainly developed in the twentieth century by Mac Lane and Eilenberg with the purpose of investigating algebraic topology, category theory has become a fast-evolving aspect of modern mathematics. In just a few decades, category theory has become the standard and formal language of homology algebra and algebraic geometry, and has contributed to many meaningful achievements, for instance in the Yoneda's Lemma and Braid groups, which serves as a neat explanation for the Yang-Baxter Equation (YBE) [5]. Moreover, category theory's main idea, which is to put mathematical structures into categories, satisfies the worldview of the Bourbaki school[5].

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

Structure of the paper

The paper mainly consists of seven sections. The Introduction section discussed the background of the research and fundamental concepts around it. The Preliminaries section, made up of two subsections, presents the pre-knowledge required to understand the paper and the notations we use. The third section defines a set of mathematical structures that will be used by the algorithm. The fourth section explains the detailed workflow procedures of the algorithm. The next section presents the conclusions of the research and future areas of improvement. Then, the algorithm is examined using several experiments and benchmarks to test its performance.

2. Preliminaries

2.1. Notations and conventions

To begin with, we use C to demonstrate a category and $\mathcal{F}$ to present a functor. $\mathcal{U}$ represents the Grothendieck universe selected and $\mathcal{F}_r$ stands for the forgetful functor. A map is denoted 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 $\mathrm{Ob}(C)$ stands for the object collection for the category C. We denote the morphism set for a category with Mor and denote the Hom–Set for a category C between elements a and b with $\mathrm{Hom}_C(a, b)$.

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, Zermelo-Fraenkel set theory (ZFC) is utilized for the set system, with Grothendieck universe concepts added [5].

We use On as the ordinal class composed by ordinals [5] such as $\{\emptyset\}$ and infinite ordinal ω, which is defined as the inductive set supported by the axiom of infinity in ZFC. Note that On is a proper class instead of a set, otherwise it will lead to the Burali-Forti paradox.

Additionally, we use S_p with S being a set and p being a statement about elements in S to represent the subset $\{x \in S | p(x) \equiv T\}$. For example, notation $\mathbb{Z}_{>1}$ represents the set of integers greater than 1. Moreover, we utilize $On_{\geq n}$ to represent ordinals not smaller than n.

2.2. Pre-knowledge

The definitions and theories presented in this paper utilize the concepts of category theory [5], ordinal theory [5] and axiomatic set theory [5]. Therefore, readers are recommended to review these concepts before going over this paper.

Axiom 2.1. *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 2.1. *The Zermelo's Well-Ordering Theorem [5].*

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

The proof can be found in citation [5].

Some preliminary knowledge also includes the Ebbinghaus Forgetting Curve equation [6] used later in this paper. This approximate function is defined below:

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

where B is percent of memorization, t is time, $c = 1.25$ and $k = 1.84$. 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 [7], 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 equation to be the following:

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

In this paper, we use notation r to represent a new random number generated, which is in reality is a pure function of the last random number, and the expression is simplified for conciseness.

3. The computerized representation of mathematical structures

Before more specific discussion, we select the Grothendieck universe $\mathcal{U}$ [5] to avoid set theory paradoxes. Next, we define meta category C where all discussions take place. There are three kinds of objects in C: symbols, notations and brackets. All symbols in C form sub-category $\mathrm{Sym}(C)$, called the symbol subcategory. All notations in C form $\mathrm{Not}(C)$ and all brackets form $\mathrm{Bra}(C)$, called the notation subcategory and the bracket subcategory, respectively. That is,

$$C = \mathrm{Sym}(C) \cup \mathrm{Not}(C) \cup \mathrm{Bra}(C). \tag{3.1}$$

The formula above specifies that the meta category C is the union set of the symbol subcategory, the notation subcategory and the bracket subcategory. It should be noted that, though we will separately manipulate on $\mathrm{Sym}(C)$, $\mathrm{Not}(C)$ and $\mathrm{Bra}(C)$ afterwards, the annotation C is still used to represent these subcategories, serving as a common context.

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

Definition 3.1. *A bracket β is defined as a set of ordered pairs in the form of below:*

$$\beta := \{(i, x)\,|i \in On_{\geq 1}\ , x \in Sym(C) \cup Not(C) \cup Bra(C)\} \tag{3.2}$$

β is valid if and only if for every element of β, its left element is unique. Next, we define filtered subsets of β. The Symbol Subset of $\beta_S := \{t \in \beta\,|t_r \in Sym(C)\}$. Similarly, the Notation Subset β_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 $\beta = \{(i, A)\}, i \in \mathbb{Z}_{\geq 1}, A \in Sym(C)$. The second type is called a Compositor where

$$\begin{aligned} \beta = \{(j, \mathcal{N})\} \cup \{(i, x)|i \in On_{\geq 1},\\ x \in Sym(C) \cup Bra(C)\}, \forall t \in \beta, j \in On_{[1,t_\ell]}\}. \end{aligned} \tag{3.3}$$

Next, we present an important definition about bracket isomorphism, which will be referred to in the third section of this paper.

Definition 3.2. *Two brackets 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 an operator (for example, + and $\cdot$) or a data type (for example, number or function). Its precise definition is as follows.

Definition 3.3. *A notation $\mathcal{N}$ is defined as an object in $Not(C)$ which is a sub-category of C. The $Ob(Not(C))$ set can be mapped to $On_{\geq 0}$ class, indicating the number of parameters that a notation takes. In which,*

$$\begin{aligned} p : Ob(Not(C)) &\to On_{\geq 0} \\ \mathcal{N} &\overset{p}{\mapsto} n. \end{aligned} \tag{3.4}$$

In that case that $n = 0$, we then 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 an infinite number of symbols as parameters, then the notation is said 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 symbols are like the content in them. A precise definition of a symbol is presented below.

Definition 3.4. *A symbol $\mathcal{S}$ is defined as an object in $Sym(C)$ which is a sub-category of C. There exists a functor from $Sym(C)$ to $Not(C)$:*

$$\begin{aligned} v : Sym(C) &\to Not(C) \\ \mathcal{S} &\overset{v}{\mapsto} v(\mathcal{S}) \end{aligned} \tag{3.5}$$

where $v(\mathcal{S})$ is the corresponding notation of $\mathcal{S}$ and must satisfy $\forall \mathcal{S} \in Sym(C), p(v(\mathcal{S})) = 0$, 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.

$$\mathcal{S} \in Ob(Sym(C)) \wedge v(\mathcal{S}) == \mathcal{N}. \tag{3.6}$$

This should not be written using $\mathcal{S} = v^{-1}(\mathcal{N})$ because, morphism v is not necessarily an injection so the existence of its inverted morphism v^{-1} cannot be guaranteed.

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 an equivalence morphism between the two in the $Bra(C)$ category, noted as below:

$$\epsilon : \alpha \equiv \beta. \tag{3.7}$$

A bracket, for simplicity, can be put into a symbol. Therefore, with these three definitions, how do we actually represent mathematical things? 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 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 situations like in the example above. To do this, we first 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 3.5. *A notation $\mathcal{N}$ is said to be inherited from notation $\mathcal{M}$ (or we say $\mathcal{N}$ is a kind of $\mathcal{M}$) when there exists a morphism between the two in $Not(C)$:*

$$h \in Mor(Not(C)) : \mathcal{M} \to \mathcal{N}. \tag{3.8}$$

It is obviously without any ambiguity that $\forall \mathcal{M}, \mathcal{N} \in Not(C), |Hom_C(\mathcal{M}, \mathcal{N})| \in \{0, 1\}$. 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 $\mathcal{M}$ and can be annotated as $\mathcal{N} \lhd |\mathcal{M}$.

Note that a symbol can be elaborated with the help of a bracket. Let $\emptyset$ be a kind of $\mathcal{I}$, representing the empty set. Then we declare $\mathcal{S}et$ as an infinite notation and significantly has $\mathcal{S}et \lhd |\emptyset$. Next, 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. Notation $\mathcal{O}$ (obviously a type) is used to declare an ordinal, which means next time if we need to use an ordinal, it just takes to instance $n \in \text{Sym}(C) \wedge v(n) == \mathcal{O}$.

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

Using the structure provided by brackets, notations and symbols, we construct the example formula below, with meta category 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: inf, int, num, $\sum$, =, ^, /, −, abs, < and for, where abs stands for the absolute value function. However, in this example, the concept of a function does not need to be defined. Additionally, as the constructed things 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 obviously int $\lhd$ |num. Table 1 demonstrates the number of parameters taken by the notations.

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

$\mathcal{N}$	int	num	inf	$\sum$	=	/	-	abs	<	for
p	0	0	0	3	2	2	2	1	2	2

Significantly, there are three types and seven operators. In the expressions, symbols are displayed in bold for recognition from ordinals, although this is not necessary. Bracket α is the result of construction. Then, just like constructing a polish notation (or prefix notation, in which construction order specifies the hierarchical relationships between the items), we construct:

$$\begin{aligned}
\mathbf{1} \in &\quad \text{Ob}(\text{Sym}(C)) \wedge v(\mathbf{1}) = \text{number} \\
\boldsymbol{\infty} \in &\quad \text{Ob}(\text{Sym}(C)) \wedge v(\boldsymbol{\infty}) = \text{infinity} \\
\boldsymbol{x} \in &\quad \text{Ob}(\text{Sym}(C)) \wedge v(\boldsymbol{x}) = \text{number} \\
\boldsymbol{n} \in &\quad \text{Ob}(\text{Sym}(C)) \wedge v(\boldsymbol{n}) = \text{integer} \lhd |\text{number}
\end{aligned}$$

$$\begin{aligned}
\alpha &= \{(0,\text{for}),(1,\beta),(2,\gamma)\}\\
\beta &= \{(0,=),(1,\delta),(2,\epsilon)\}\\
\gamma &= \{(0,<),(1,\zeta),(2,\mathbf{1})\}\\
\delta &= \{(0,\sum),(1,\eta),(2,\boldsymbol{\infty}),(3,\theta)\}\\
\epsilon &= \{(0,/),(1,\mathbf{1}),(2,\iota)\}\\
\zeta &= \{(0,\text{abs}),(1,\boldsymbol{x})\}\\
\eta &= \{(0,=),(1,\boldsymbol{n}),(2,\mathbf{1})\}\\
\theta &= \{(0,\wedge),(1,\boldsymbol{x}),(2,\boldsymbol{n})\}\\
\iota &= \{(0,-),(1,\mathbf{1}),(2,\boldsymbol{x})\}.
\end{aligned}$$

We can combine the simple brackets together and form larger brackets in the form shown below. Declaration of the four symbols are omitted.

$$\begin{aligned}
\alpha = \{(0,\text{for}),(1,\beta),(2,\{(0,<),(1,\{(0,\text{abs}),\\
(1,\boldsymbol{x})\}),(2,\mathbf{1})\})\}\\
\beta = \{(0,=),(1,\delta),(2,\{(0,/),(1,\mathbf{1}),(2,\{(0,-),\\
(1,\mathbf{1}),(2,\boldsymbol{x})\})\})\}\\
\delta = \{(0,\sum),(1,\{(0,=),(1,\boldsymbol{n}),(2,\mathbf{1})\}),(2,\boldsymbol{\infty}),(3,\\
\{(0,\wedge),(1,\boldsymbol{x}),(2,\boldsymbol{n})\})\}.
\end{aligned} \tag{3.9}$$

It is still possible to combine these all into a large bracket definition for α, but in that way it may be even harder for people to understand, although it makes no difference for algorithms and programs.

Another issue is in regard to the recognition between 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 statements (for instance, $\{(0,=),(1,x),(2,y)\}$ or the expression for the Pythagorean theorem). In this system of representation, the second sort of bracket is called a micro-statement, whose precise definition is presented below.

Definition 3.6. *A micro-statement is defined as an object in Bra(C) whose notation $\mathcal{N}$ has $p(\mathcal{N}) = 2$ and can be evaluated into a Boolean.*

It is essential to pay attention to the fact that a bracket with 2-parameter notation is not necessarily a micro-statement. This conclusion can be illustrated via the example of the bracket $\{(0,+),(1,x),(2,y)\}$*, whose notation takes 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 is that the evaluation process is not declared inside the representation system, whose definition comes below. All micro-statement brackets form the micro-statement sub-category of Bra(C) and is annotated as Mic(C).*

Briefly, an evaluation can be understood by imagining a map which takes a micro-statement bracket and sends a Boolean value as output. The precise definition is present below.

Definition 3.7. *An evaluation is a morphism from Mic(C) to a Boolean algebra $\mathcal{B}$ with the form below.*

$$\begin{gathered} eva\ell : Ob(Mic(\mathcal{C})) \to \mathcal{B} \\ m \overset{eva\ell}{\mapsto} eva\ell(m). \end{gathered} \tag{3.10}$$

The Boolean value $eva\ell(m) \in \{0, 1\}$ is called the evaluation result of a micro-statement. If for bracket m there 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 fourth section of this paper in the discussion on how to manipulate the data structures described in this section for derivation later.

4. 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 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 and elementary that,

$$\forall \alpha, \beta \in \mathrm{Ob}(\mathrm{Mic}(\mathcal{C})), \forall \left|\mathrm{Hom}_{\mathrm{Mic}(\mathcal{C})}(\alpha, \beta)\right| = 1. \tag{4.1}$$

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

$$\mathcal{D} := \{(i, \mathcal{M}_i) \,|\, i \in On_{\geq 1}\} \tag{4.2}$$

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. The content of the initial pool is set to the condition, i.e., 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 $\mathcal{R}$ (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 the derivation, illustrating how the algorithm works from the ground up is defined below.

Definition 4.2. *A step of derivation is a map existing in the derivation set $\mathcal{D}$, mapping from a pair to another, whose ordinal increases by* 1*. This map is annotated using the script word step. The form is shown below:*

$$(i, \mathcal{M}_i) \overset{step}{\mapsto} (i + 1, \mathcal{M}_{i+1}). \tag{4.3}$$

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,

$$\mathcal{M}_{i+1} = \mathcal{F}_i \mathcal{M}_i. \tag{4.4}$$

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

Definition 4.3. *A derivation set is thought to be successful when the condition below is satisfied when the condition below is true:*

$$\exists n \in [0, \omega), \mathcal{R} \subseteq Ob\left(\mathcal{M}_n\right). \tag{4.5}$$

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, and cannot be judged before execution. Next, we investigate the execution of the recurse functor, which is 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 if no global variable. To solve this problem, we utilize the r random number function utility described in the Preliminary section.

Definition 4.4. *A reflection functor* $\mathcal{K}$ *is defined as an functor object in a functor category* $\mathcal{R}ef$ *that maps between two micro-statement pools. Because of Zermelo's well-ordering theorem we mentioned in the Preliminary section,* $Ob(\mathcal{R}ef)$ *can be put into well-order, with ordinal i. All of the reflection functors form a functor category* $\mathcal{Q}$*, called the reflection knowledge base, or for short the knowledge base. The detailed explanation and description, including the morphisms inside the knowledge base, will be discussed later in this section of the paper. We annotate each of the functors 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,*

$$\begin{aligned} \sigma &= T(\mathcal{K})_\ell \\ \tau &= T(\mathcal{K})_r. \end{aligned} \tag{4.6}$$

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 is called an isomorphism inspection and is described as below.

Definition 4.5. *The object 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.*

$$Ts{:=}\left\{t \in \mathcal{M}_i | t \simeq T(\mathcal{K})_\ell\right\}. \tag{4.7}$$

What is done next is called a transformation. Foremost, a set of transformation rules are created in order to establish bijections from the micro-statement, which is about to be manipulated, the left of the template pair σ. Secondly, the reverse map is applied to the right of the template pair τ and then the transformation process execution is done. Next, we delineate the definition of transformation mapping.

Definition 4.6. *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,*

$$\begin{gathered}\mathcal{T}: \alpha_{\mathcal{B}} \cup \alpha_{\mathcal{S}} \to \beta_{\mathcal{B}} \cup \beta_{\mathcal{S}} \\ \mathcal{S}_1 \overset{T}{\mapsto} \mathcal{S}_2.\end{gathered} \tag{4.8}$$

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

$$(\mu')_{\mathcal{B}} \cup (\mu')_{\mathcal{S}} = \mathcal{T}^{-1}\left(\mu_{\mathcal{B}} \cup \mu_{\mathcal{S}}\right). \tag{4.9}$$

And then micro-statement μ' is the freshly baked result of derivation. 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, the derivation is to be done in practice using recursive functor $\mathcal{F}_i$, which selects $\mathcal{K}$ as the reflection functor to utilize, following the specification we present below.

The recursive functor $\mathcal{F}_i$ decides the reflection functor to use based on these criteria: The random seed denoted with r, the knowledge base that is the functor category that all reflection functors form, as well as an integer j, indicating the last time $\mathcal{K}_j$ is executed. Notice that during each time of execution, r is calculated by the random generator function and passed as a parameter implicitly, which guarantees F_i's being well-defined, meaning that the result of F_i is certain for each input. To begin with, we will investigate the organization of objects in the knowledge base Q.

Definition 4.7. *The initial map is the original state of the knowledge base Q when Q is created. Initially, after the reflection functors are sorted, while discussions around this take place before, they form a circular path with connections of morphisms. That is, for $N = |Ob(Q)|$,*

$$\begin{gathered}\mathcal{K}_n \xrightarrow{k_n} \mathcal{K}_{n+1}, n \in \mathbb{Z}_{[0,N-1]} \\ \mathcal{K}_N \xrightarrow{k_N} \mathcal{K}_0.\end{gathered} \tag{4.10}$$

Graphically, Figure 1 illustrates the initial map.

$$\begin{tikzcd}
 & \mathcal{K}_0 \arrow[dl, "k_0"] & \\
\mathcal{K}_1 \arrow[d, "K_1"] & & \mathcal{K}_N \arrow[ul, dashed, "k_N"'] \\
\mathcal{K}_2 \arrow[rr, "k_2"] & & \mathcal{K}_3 \arrow[u, dashed, "K_3"]
\end{tikzcd}$$

Figure 1. The initial map.

A morphism in this category means that if the reflection functor on its start is executed, then the target functor will be executed next time. That is, to be more specific, if a functor in 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, it is ordinary. The process of determining the next functor of an extroverted functor is 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 we say this new method of implementing a prover makes the deduction algorithm low-knowledge, it turns out to have an infrastructure called the knowledge base. This is not conflicting since the knowledge we refer to in the Introduction section means the algorithm is automated, meaning it does not require users to provide much input and it does not know exactly what is under manipulation. On the other hand, however, the word knowledge here means the storage of reflections, which can be rules, axioms, theorems, lemmas and even strategies.

Definition 4.8. *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 Q, right from starting 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 an introverted or ordinary functor $\mathcal{K}_j$ then significantly,*

$$w = k_j\left(\mathcal{K}_j\right). \tag{4.11}$$

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

$$w = (r \bmod \sum_{u=0}^{N}) \left|Hom_Q\left(\mathcal{K}_j, \mathcal{K}_q\right)\right|. \tag{4.12}$$

It is easy to understand that for each reflection functor $\mathcal{K}_a$ and $\mathcal{K}_b$, the count of set $Hom_Q\left(\mathcal{K}_a, \mathcal{K}_b\right)$ 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 discussed in Andrew S. Tanenbaum's Operating Systems: Design and Implementation [8].

An issue that is worth attention is that of the random generator defined in the following recursion formula: [7]

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

Then each r must belong to region $\mathbb{Z}_{[0,2^{32})}$. That is, obviously, if the count of all morphisms is more than 2^{32}, then some of the functors that 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 gigabytes and will not be exceeded in most engineering situations. Obviously when using the portable recursion equation, the quality of the random numbers generated may be lower than some technical implementations, it can be conveniently expressed in this paper. If the algorithm is implemented in the real-world, a randomness collector, which generates random numbers from the real world, for example, the vibration of the computer motherboard, will probably act as a better solution.

The more detailed explanation of the recursive functor can be found in the algorithm below (Algorithm 1).

Algorithm 1 Pseudo code for recursive functor $\mathcal{F}_t$

```
Require: Pool $\mathcal{M}_t$, Knowledge base $Q$, random $r$, Just used reflection functor index $i$
Ensure: Next pool $\mathcal{M}_{t+1}$
function $\mathcal{F}_t(\mathcal{M}_t, Q, r, i)$                    ▷ This is the sub-process
    $\mathcal{K} \leftarrow$ SelectReflectionFunctor($Q, r, i$)
    return $\mathcal{K}\mathcal{M}_t$                                  ▷ Apply functor $\mathcal{K}$ to pool
end function
function SelectReflectionFunctor($Q, r, i$)                 ▷ Just used: $\mathcal{K}_i \in \mathrm{Ob}(Q)$
    $M \leftarrow Mor(Q)$
    for all $m \in M$ do                                 ▷ Get morphisms starting at $\mathcal{K}_i$
        if $s(m)! = \mathcal{K}_i$ then
            $M \leftarrow M \setminus \{m\}$
        end if
    end for
    if $|M| == 1$ then                                   ▷ The easiest condition
        return $t(m \in M)$
    else
        $n \leftarrow r\%|M|$
        return $t(n)$
    end if
end function
```

Note that we use the statement $\mathcal{K}\mathcal{M}_t$ to represent applying functor $\mathcal{K}$ to category $\mathcal{M}_t$ and get the result category. The detailed steps of the functor execution are defined previously.

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

Definition 4.9. *The process of knowledge reinforcement is performed 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. The next time a proof is issued, the modified knowledge base is loaded into an instance and acts 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 Figure 2.

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

Figure 2. An example map after several reinforcements.

Readers may notify that that $\mathcal{K}_1$ has a morphism targeting $\mathcal{K}_N$ and $\mathcal{K}_N$ also has a morphism targeting $\mathcal{K}_1$. This consequence is normal and natural following the evolution. Another possible issue is that, after some recursions, there may exist things whose weight is below the least acceptable numerical, that is, they are completely forgotten. Well, in such a case, there will be absolutely no recursion involving them. The only way to restore their activity is to reconfigure their parameter manually.

To summarize, we can illustrate the entire process of the workflow of the algorithm with the following diagram on the next page (Figure 3).

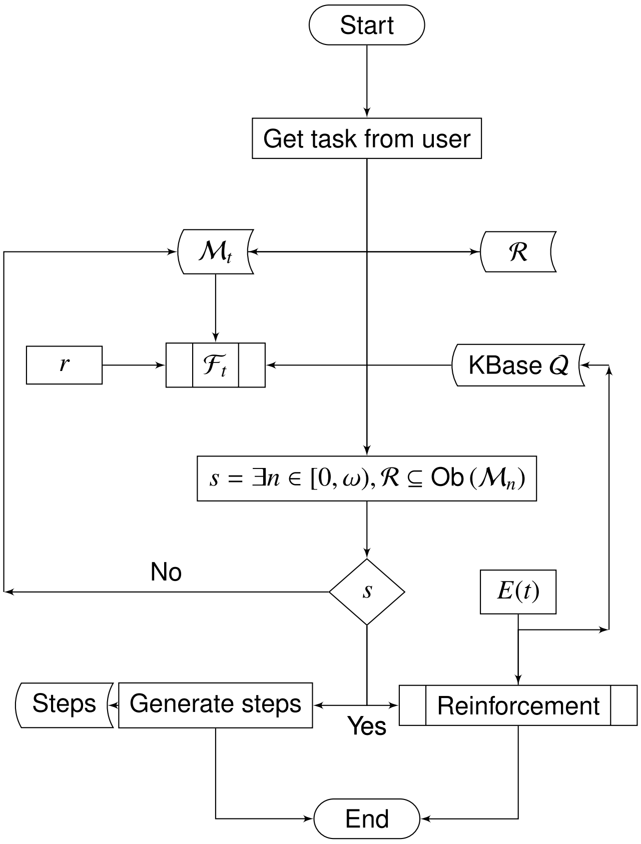

Figure 3. The flowchart of the entire algorithm.

Via the process we described in the fourth 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.

5. Testing the algorithm's performance through benchmarks

The discussion below is based on an implementation of the algorithm by us, called DefQed, and is open-sourced on [9]. It will be further described in the sixth section. To illustrate the performance of the brand-new algorithm, we construct a relatively simple exemplification. The source code describing the demonstration can be found on the internet through [10]. The code for Wolfram Engine (or Wolfram Mathematica) is also presented. The timing cost by the three provers are shown in Table 2.

Table 2. Timing (ms) of the three algorithms.

Round♯	DefQed	Wolfram Engine
1	576	141
2	727	250
3	752	375
4	889	266

The data can be further illustrated in Figure 4. The blue line stands for the time (milliseconds) of DefQed, while the orange line stands for the time (milliseconds) of Wolfram Engine.

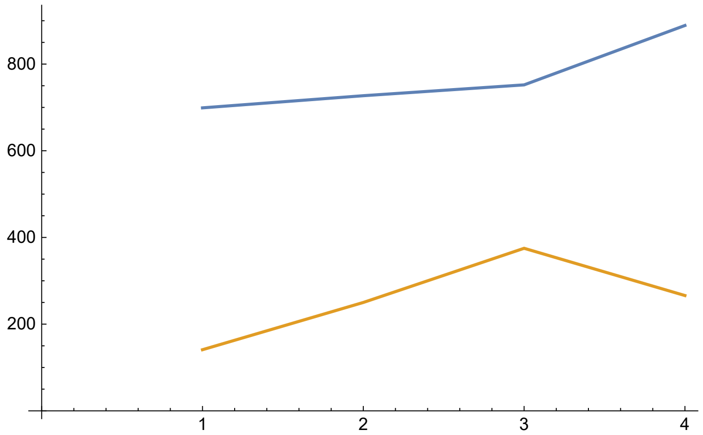

Figure 4. The plots of the data measures.

From Figure 4 we can obviously tell the time cost of the reasoning increases by the round, as more reflections are inserted into the database. It should be noted that, for time reasons, the reduction algorithm is not implemented in this version of DefQed, and the coding quality can also be further improved.

The experiment is done on a Hewlett-Packard ZBook mobile workstation 14u G5 [11]. During the experiment, all the programs/applications not necessary are closed except for the terminal emulator (ConEmu) and the program to benchmark. The operating system is Microsoft Windows 11 Pro for Workstations Insider Preview (Build 25211). The processor is Intel Core i5 8250U and the memory size is 8.0 GB. We used Wolfram Language version 12.0 and DefQed version 0.03. As DefQed utilizes MySQL as its database, it is necessary to point out we are using MySQL version 8.0 (Community). The DefQed source code is built with option 'Release Mode' for Microsoft Windows x64 processor with NET 6.0 platform.

We also need to point out that, though in each round DefQed seems to be slower than others, it does not mean that the algorithm has a worse efficiency and performance when comparing to others. The performance difference between platform and programming language utilized by the two, NET and mainly C contributes to the difference of the two applications, not mentioning that while Wolfram Engine can only prove equations, DefQed is a generic approach. The quality of the optimization of the codes are also not on be same basis.

6. Conclusions and future work

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

Currently, we 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 [9]. It should be noted that the version is very early and some aspects of the algorithm we presented in the paper are not implemented.

Future works are the followings below:

- **Continuing the implementation of the corresponding software.** Currently, the software we developed is still under early development and a great many of the aspects we presented in the paper above, including the self-evolution logic, have not been constructed yet.
- **Improving the algorithm.** The algorithm we 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 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.

Use of AI tools declaration

The authors declare they have not used Artificial Intelligence (AI) tools in the creation of this article.

Acknowledgments

I express my sincerest gratitude to my tutor Prof. Shao Xinhui, the Yingcai Project of China, which enables senior high school students to conduct science research, my senior high academy Northeast Yucai School (NEYC) of China and the corresponding university Northern Eastern University (NEU) of China. I also thank my parents for encouraging me to conduct this research project. Additionally, I had little knowledge about LaTeXbefore writing this paper. The *lshort* documentation and the TeXStack Exchange helped me a lot. Here I express my genuine thankfulness to the online resources and documentation of the used LaTeXpackages and software components.

This research received general aid from the Yingcai Project of China.

Conflict of interest

The authors declare no conflict of interest.

References

1. S. Wolfram, *Wolfram language & system documentation center*, Wolfram Research Inc., Champaign, IL, US, 12Eds., 2019.

2. X. S. Gao, *Geoexpert*, Beijing, 1998. Available from: `http://www.mmrc.iss.ac.cn/gex/`.

3. L. D. Moura, S. Ullrich, The lean 4 theorem prover and programming language, *Autom. Deduction -CADE*, 2021, 625–235. http://doi.org/10.1007/978-3-030-79876-5˙37

4. L. D. Moura, N. Bjørner, *Z3: An efficient smt solver,* In: Tools and Algorithms for the Construction and Analysis of Systems, **4963** (2008), 337–340. http://doi.org/10.1007/978-3-540-78800-3_24

5. W. W. Li, *Daishuxue Fangfa*, 1 Ed, High Education Press, Beijing, 2019. ISBN 978-7-04-050725-6.

6. H. Ebbinghaus, *Memory: A contribution to experimental psychology*, Teachers College, Columbia University, New York City, 1913. Available from: `https://archive.org/details/memorycontributi00ebbiuoft/page/n5/mode/2up`.

7. K. Entacher, *A collection of selected pseudorandom number generators with linear structures*, 1997. Available from: `https://www.semanticscholar.org/paper/6ae5fe88d296e70f188b0d12207d468c8f36e262`.

8. A. S. Tanenbaum, *Operating systems: Design and implementation (volume 1)*, Publishing House of Electronics Industry, Beijing, 3 Eds., 2015. ISBN 978-7-121-26193-0.

9. Z. J. Wang, *The defqed repository*, 2022. Available from: `https://github.com/ZijianFelixWang/DefQed`.

10. Z. J. Wang, *Benchmark of the defqed algorithm*, 2022. Available from: `https://github.com/ZijianFelixWang/DefQed-Benchmark`.

11. H. Packard, *The hp zbook 14u g5 specifications*, 2018. `https://support.hp.com/us-en/document/c05873311`.