CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Learning the intrinsic value of theorems

Estimating usefulness of theorems with neural networks

Master's thesis in Computer science and engineering

Johan Vallander

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Learning the intrinsic value of theorems

Estimating usefulness of theorems with neural networks

Johan Vallander

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Learning the intrinsic value of theorems
Estimating usefulness of theorems with neural networks
Johan Vallander

Supervisor: Sólrún Einarsdóttir, Department of Computer Science and Engineering
Examiner: Moa Johansson, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Learning the intrinsic value of theorems
Estimating usefulness of theorems with neural networks
JOHAN VALLANDER
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

We investigate whether neural networks can learn some notion of usefulness/interestingness of theorems, or as we chose to call it "intrinsic value". This we define as the ability to take part in the deduction of other (sufficiently "beautiful") theorems. To study this we first devise a definition of what constitutes a "beautiful" theorem. We then construct a symbolic system which, starting from a set of axioms, randomly deduces new theorems from existing ones. Using this symbolic system we gather a lot of beautiful theorems, and consequently theorems with intrinsic value. We experiment with different metrics of intrinsic value to find out which works best. We then use this metric together with the collected theorems to train neural networks to classify a theorem as useful or not. We find, using MPNN and DAG-LSTM architectures, that this is possible. We also find that we can optimize the discovery of beautiful theorems with the aid of these trained neural networks.

Keywords: Computer science, interestingness, usefulness, theorems, automatic deduction, project, thesis, dag-lstm, mpnn

Acknowledgements

Thanks to Sólrún Einarsdóttir for all support, and thanks to Nicholas Smallbone for his knowledge and insights. Lastly thanks to Moa Johansson for trying to persuade me from constructing my own system for automated deduction - I should probably have listened. Johan Vallander, Gothenburg, 2026-01-18

Contents

List of Figures

List of Tables

1

Introduction

Not all discoveries in logic or mathematics are made on purpose by someone trying to prove that very thing. Sometimes mathematical discoveries happen by chance, by manipulation of formulas which stumble upon an interesting conclusion or a novel way of seeing something. This is the inspiration for this thesis, where we try to build a neurosymbolic system that to some degree replicates this discovery process.

To this day, the introduction of machine learning towards the field of mathematical and logical reasoning has mainly focused on teaching neural networks to prove theorems. Examples of this include The deepSeek-Prover [1] by Xin et al. and the self-reinforcement system Minimo [2] which teaches itself both to pose increasingly challenging problems and to prove these problems. We choose to take a different strategy, which, to the best of our knowledge is a novel one. Our approach is aimed toward teaching a neural network to recognize which theorems are interesting/useful, or as we chose to call it, have "intrinsic value", while leaving their generation to symbolic stochastic methods.
Neural networks have recently shown a surprising ability to perform tasks previously beyond the scope of computers. They can solve problems whose solutions are hard to define and whose heuristics are therefore way too complicated to be specified in an algorithm. Such tasks include problems such as differentiating between a picture of a cat or a dog, different styles of music, or sentences that carry a positive or negative sentiment. It is the question of this thesis whether a neural network can also be taught to discriminate between intrinsically valuable theorems and more useless ones, even if it is hard to strictly define what constitutes such a theorem.
Practically we intend to create a symbolic system that performs a Monte Carlo simulation of theorem deduction, thereby providing training data as positive and negative examples, consisting of theorems that have taken part in the deduction of beautiful theorems, and those that did not. This data will be used to train neural networks to classify theorems as valuable or not. By coupling the neural network to the symbolic system in a self reinforcing loop (reinforcement learning), this could serve as a theory explorer of arbitrary logic. Such coupling does, however, fall outside the scope of this work.

Specifically, we intend to:

- Create a symbolic environment which will randomly infer new theorems from existing ones.
- Use the symbolic environment to classify theorems as intrinsically valuable or not and gather a lot of training data.
- Implement graph neural networks (GNNs) capable of classifying theorems as valuable or not.
- Feed the gathered examples as training data to the GNNs.
- Couple the trained GNNs with the symbolic environment to evaluate the gain from the predictions of the GNNs.

The research hypothesis, whether a neural network can capture the notion of a valuable theorem, will be evaluated based on how well it learns to classify theorems as intrinsically valuable or not, as well as how much such a trained neural network aids the symbolic environment in the discovery of new theorems.

2

Background

Although our approach is novel, the concept of intrinsic value has been previously elaborated upon in the literature. Bengio and Malkin discuss, in their paper "Machine learning and information theory concepts toward an AI mathematician" [3], what makes a theorem useful. They base their reasoning on concepts from information theory and define "usefulness", or "interestingness", as the ability of a theorem to compress the size of the proofs of all provable theorems. We will make an attempt to somewhat follow this definition in our definition for "intrinsic value" presented later on.

In the field of automatic conjecturing we have systems such as QuickSpec [4] and RoughSpec [5]. In these systems equalities between formulas are found by evaluating different formulas while comparing the outcome to other formulas. For example, given a sorting function $sort()$ taking a list x the algorithm will find, through testing, that $sort(x) == sort(sort(x))$. But it also holds that $sort(sort(x)) == sort(sort(sort(x)))$ and so on, so these systems naturally must deal with the question of what constitutes an interesting finding worth presenting to the user. One major strategy employed by these systems is to discard any property which follows from equational reasoning from previously discovered properties. For example $sort(sort(x)) == sort(sort(sort(x)))$ follows from $sort(x) == sort(sort(x))$, so the user is spared this redundant piece of information. Our goal differs from these systems in that we are not aiming to create any comprehensible overview of some underlying logic for a human, but instead find the theorems which are most useful in creating other theorems, or as Bengio and Malkin phrase it, compress the size of all proofs. We do however need some sensible limitation on what is worth proving (since the amount of possible theorems is infinite) and we will for this aim employ similar, but less stringent, measures as employed in QuickSpec and RoughSpec.

The idea of estimating this "intrinsic value" is as previously stated an untested endeavor. Neural networks have, however, successfully been employed toward a similar task when it comes to theorem proving. Among many others, Crouse et al. [6] have utilized neural networks for premise selection (choosing which known theorems to serve as the beginning step of the proof), and proof step classifications. So even though the general usefulness of a theorem has never previously been subject to the evaluation by neural networks, the classification of the usefulness of a theorem

toward a specific problem has been proven successful many times. This work will also follow the strategy of Crouse et al. by representing formulas as Directed Acyclic Graphs (DAGs) and processing them using graph neural networks (GNNs).

3

Technical background

3.1 Metamath

Metamath [7] is a small language for writing and verifying proofs. When implementing theorem representation and inference rules, we have chosen to follow the abstract syntax of this language. Our implementation thus rests on the firm foundation that this recognized language has to offer. Metamath contains only a minor set of constructs, yet it is capable of expressing almost any logic. Most commonly, the language is used together with the library *set.mm* which encodes most of what can be expressed within ZFC set theory, but other libraries also exist, such as *peano.mm* that specifies a formulation of Peano arithmetic. The two systems of logic which are explored in this work also use subsets of axioms from these two libraries.

Metamath is a language that describes valid substitutions of strings. A proof in Metamath is a statement of symbols followed by the "commands" that creates that specific sequence, thereby proving the statement. Each command is a reference to earlier proofs or specified axioms, which works by pulling and pushing symbol sequences (strings) to a stack. For example, *set.mm* specifies the axiom *wi*, stating that $\phi \implies \psi$ is a well formed formula (*wff*).

```
wi $a wff ( ph -> ps ) $.
```

ϕ and ψ, indicated by "ph" and "ps" respectively, are variables previously declared to be of type "wff", and match with sequences on the stack of the same type ("wff"). Effectively, this means that any two sequences of type wff, i.e previously proven to be well-formed formulas, can be combined into a new sequence of type "wff" by substituting the first sequence with ph and the second with ps. Similarly, there are other axioms for other logical connectives providing us with a recursive way to produce a specific string. We do not adhere to Metamath's string-based approach and instead choose to work with formulas encoded as graphs, namely as abstract syntax trees (AST). Where Metamath's invocation of the axiom mentioned above *wi* consumes two entries from the stack in order to produce a new one, our system instead represents an implication as a node with two ordered children. This graph structure is logically equivalent, but offers a more comfortable representation.

3.2 Inference rules

Central to logic is the concept of *inference methods* which means, as the name implies, methods to infer new theorems from known ones. There are different philosophies when it comes to choosing inference methods for a logical system. A Hilbert-type system for example keeps the amount of inference methods to a minimum while compensating with an extender number of axioms. The most extreme types of Hilbert systems implement only *Modus Ponens*, and in the case of predicate logic, *For-all introduction*. On the other end of the spectrum we have systems like natural deduction, where there are instead many inference methods and fewer axioms. For purposes such as automated theorem proving, Hilbert style systems benefit from ease of implementation. In our project we have chosen to limit ourselves to four inference methods: modus ponens, for-all introduction, for-all-elimination and exist-introduction. These will each be broadly described hereafter.

3.2.1 Modus Ponens

The inference rule modus ponens, formally presented as $\phi \implies \psi, \phi \vdash \psi$, states that if we have a theorem of the form $\phi \implies \psi$, where the *antecedent* ϕ and *consequent* ψ are any well-formed formula, and we also have a theorem ϕ, we can form a new theorem ψ.
Modus ponens allows much of the functionality present in e.g natural deduction to be encoded as axioms. For example given ψ and ϕ natural deduction allows the inference of $\psi \wedge \phi$, called *and-introduction*. Using Hilbert style there could instead be an axiom stating that $\psi \implies (\phi \implies (\psi \wedge \phi))$ allowing the and-introduction to be performed in two steps.

3.2.2 Substitution

The rule of substitution simply states that if we know that two things are equal, $a = b$, then we replace any occurrences of a in any formula with b, and vice versa, thereby inferring a new theorem.

3.2.3 For-all introduction

This inference method, also called "universal introduction", allows us to introduce a $\forall$ quantifier in a formula. It states that if a formula $f(a)$ holds for an arbitrary term a, it also holds for all possible terms, i.e $\forall x f(x)$.

3.2.4 For-all elimination

For-all elimination allows us to go the other way and remove a universal quantifier from a formula. The rule says that if a formula holds for any term, i.e $\forall x f(x)$, then we can infer that it holds for an arbitrary term as well, i.e $f(a)$

3.2.5 Exist introduction

The rule for introducing an existential quantifier says that if a formula holds for a specific a, i.e $f(a)$, then we can conclude there exists a term for which f holds, i.e $\exists x f(x)$

3.3 Generalizations and instances

Consider the propositional formula $\phi \vee \neg\phi$. This tautology says that something, ϕ, is either true or false. It doesn't matter what we say ϕ is. Setting $\phi = \psi \wedge \chi$ and thereby creating the formula $(\psi \wedge \chi) \vee \neg(\psi \wedge \chi)$ does not change the truth value of the tautology since $(\psi \wedge \chi) \vee \neg(\psi \wedge \chi)$ is implied by $\phi \vee \neg\phi$. We say that $\phi \vee \neg\phi$ is a *generalization* of $(\psi \wedge \chi) \vee \neg(\psi \wedge \chi)$ while the latter is called an *instance* of the former.

3.4 Unification

Unification is the process of finding matching instances of two formulas which enables e.g modus ponens. Consider for example the implication $(\phi \to (\phi \to \chi)) \to (\phi \to \chi)$. Its antecedent is $\phi \to (\phi \to \chi)$ which can be satisfied by the theorem $\tau \to (\upsilon \to (\tau \wedge \upsilon))$ by the process of unification. By setting $\phi = \tau = \upsilon$ and $\chi = (\tau \wedge \upsilon)$ the two theorems matches The consequent of the implication, $\phi \to \chi$ then gives us a new theorem $\phi \to (\phi \wedge \phi)$.

3.5 Pre-filtering

The process of checking if a formula is a generalization or instance of another, or if they are unifiable, is a rather complex operation which amounts to a major performance bottleneck when dealing with many theorems. Automated theorem provers normally employs sophisticated indexing methods which allows for more or less efficient pre-filtering of e.g possible unification candidates. One of the simpler pre-filtering techniques is called path-indexing [8] and works by referencing a theorem to all the *paths* through the abstract syntax tree. As an example, the theorem $\phi \vee \neg\phi$ carries the paths $\langle\vee\rangle$, $\langle\vee, 1, \phi\rangle$, $\langle\vee, 2, \neg\rangle$ and $\langle\vee, 2, \neg, 1, \phi\rangle$, where the numbers indicate if the path went through the first or second child of a node.

3.6 Flat-terms

Although this work mainly treats formulas as graphs (ASTs), the performance critical process of unification benefits from dealing with flat-terms in order to avoid recursion which is very slow in Python. Flat-terms is a way to represent a formula one-dimensionally. It is efficient for machines but sligthly tricky for humans, especially when dealing with long formulas. It works by placing the operator first followed by its arguments. The flat-term representation of $\phi \vee \neg\phi$ is for example

$\vee\phi\neg\phi$ while $(\phi \to (\phi \to \chi)) \to (\phi \to \chi)$ becomes $\to\to \phi \to \phi\chi \to \phi\chi$. Using this representation unification can be done in a for loop instead of by using recursion.

3.7 Neural networks

3.7.1 Graph neural networks (GNN)

Many kinds of data can be represented as graphs, i.e nodes connected by edges. Examples of this includes molecules, road networks and social networks. Neural networks designed to deal with these kinds of irregular data structures are called graph neural networks (GNN) [9]. They come in many flavours of which a few examples are Message passing neural network (MPNN), Graph convolutional networks (GCN) or Graph attention networks (GAT). All GNN architectures are built upon the foundation of message passing. The nodes and/or edges begins with some initial representations which are then used to create messages to the neighbours of the node. Each node then aggregates, often by taking the sum, max or mean value, of all incoming messages and use this aggregated value to create new representation for the nodes and/or edges. This step of aggregating information from the neighbours of the nodes is referred to as a layer of the GNN. A model usually consists of several layers, allowing a node to be influenced not only by its immediate neighbours but also from nodes several steps away. The layers are often identical and share weights but they don't have to be.
When a sufficient number of aggregations has occurred, the updated node features can be used as they are, representing some aggregated information about that node. This can be used to predict properties about this node. For example, if a node represents a person in a social network, with edges to other people, the feature representation of that node could be used to predict interests, based on the interests of the person's friend. This is called "node prediction". Sometimes however, as is the case in this work, we want to make predictions about the properties of the whole graph from the individual node representations. This is called *pooling*. A simple and common pooling method, used in this work, is called *max pooling* where each feature (i.e number in a vector) is derived by finding the maximum value of that feature among the nodes.

3.7.2 LSTM and Tree LSTM

The "Long Short Term Memory" (LSTM) [10] is a recurrent neural architecture used to process one-dimensional sequential data, for example human written text. It is aimed toward mitigating the vanishing gradient problem by incorporating memory cells and having trainable gates that learn what to store and what to forget. There exists a generalization of this architecture towards tree-structured data called Tree LSTM [11]. Where the LSTM can only draw information from the previous step/node, the Tree LSTM can incorporate information from several child nodes.

3.8 Robinson arithmetics

Robinson arithmetic is a subset of Peano arithmetic [12] which lacks the induction axiom. It is an axiomatic formalization of the positive integers. It is based on the term 0 and a successor function $S(t) = 1 + t$. The number 1 is represented as $S(0)$, 2 is represented as $S(S(0))$ and so on. The axioms of Robinson arithmetics are specified in A.2

3.9 Monte Carlo simulations

A Monte Carlo simulations [13], or Monte Carlo methods, are algorithms to estimate probabilities by random sampling, usually implemented with computers. They are often used when a problem is too complex to calculate the probabilities with mathematics. The idea is to repeatedly carry out a stochastic simulation and let the results be an estimate of the probabilities for certain outcomes. For example, throwing five dice a million times and counting how many times a full house appears gives a good estimate of the probability of a full house in for example Yatzy or other dice games.

4

Methods

4.1 Intrinsic value and beautiful theorems

Bengio and Malkin define "usefulness" or "interestingness" [3] as the ability for a theorem to compress the proof space of all provable theorems. In other words, a theorem is considered useful if it can make the proofs of other theorems shorter. In our Monte Carlo simulation, where theorems are deduced randomly, this translates to the theorem being more likely to take part in the deduction of other theorems. This is the core idea behind our definition of intrinsic value. However, since the number of possible theorems is infinite, and therefore beyond the scope of a Monte Carlo simulation, we have limited ourselves to what we classify as "beautiful" theorems. What makes a theorem beautiful is, of course, a subjective question without any clear answer. We do, however, believe that our attempts to define this formally capture this undefinable quantity to at least some degree. Mainly, we consider that a beautiful theorem is:

- Easily comprehensible, i.e. short enough.
- Unique, in whole and in all of its constituents.
- Includes many different operators or types of terms

The first point is easily formalized as the length of the formula being within a specified limit.
The second requirement of uniqueness is implemented as a search for generalizations among previously discovered theorems. If the theorem or any of its parts are already known, the theorem is not considered beautiful. For example, $\phi \to (\psi \vee \neg\psi)$ is not considered beautiful if $\psi \vee \neg\psi$ is already known, neither is $(\phi \wedge \chi) \vee \neg(\phi \wedge \chi)$ since the latter is a generalization of the former and contains no new unique information. This definition of beauty is used for propositional logic. With the introduction of equality in our logic, this definition is made more stringent by the additional requirement that the smallest possible representation of a theorem, according to the already deduced equalities, must be unique. For example, if the system already knows that $t + 0 = t$, then $2 + 2 + 0 = 4$ is not unique if $2 + 2 = 4$ is already known, because $2 + 0$ can be shortened to 2 using the known equality $t + 0 = t$. This serves the same purpose and works similarly to the pruning technique used by Smallbone et. al in their implementation of QuickSpec [4], something which is discussed more thoroughly in the section on Theorem shortening and sorting.

Our third qualification for beauty is based on the idea that a theorem that combines different operators and different kinds of symbols is more beautiful than a theorem with only similar kinds of symbols. According to a poll in the Fall 1988 Mathematical Intelligencer(vol. 10, no. 4) the most beautiful formula existing is Euler's identity [14] $e^{\pi i} = -1$. Zeki et. al mentions in the article "The experience of mathematical beauty and its neural correlates" [15], that this formula combines the symbols e, π, i and the number 1 with the operators for power, multiplication and subtraction. According to our last qualification Euler's identity is more beautiful than something of the same size involving fewer operators and symbols, such as $2 + 2 + 2 = 6$.
Given the requirement for uniqueness is fulfilled, a theorem T is given a preliminary "beauty score" $b'(T)$ based on the other two qualifications according to the formula $b'(T) = \frac{|\text{Unique symbols in T}|}{|\text{symbols in T}|}$ where every variable term $(\phi, \psi, \chi, r, s, t, x, y, z)$ is considered identical. If this beauty score is above a certain threshold K the theorem is considered to be beautiful. In summary, we have that the beauty $b(T)$ of theorem T is described by:

$$b(T) = \begin{cases} 0, & \text{If the theorem or any of its constituents are known} \\ 0, & \text{If b'(T)} < \text{K} \\ b'(T), & \text{Otherwise} \end{cases}$$

Theorems with intrinsic value are those that have taken part in the deduction of beautiful theorems. The intrinsic value is a specific value that is accumulated recursively such that a beautiful theorem adds its calculated beauty, to its direct parents, which in turn does the same to its parents and so on, for every step decayed by an *alpha* constant, in a fashion similar to Q-learning. Formally it is described by the following formula:

$$I(T) = \alpha * \Sigma c \in children(T) B(c) + I(c)$$

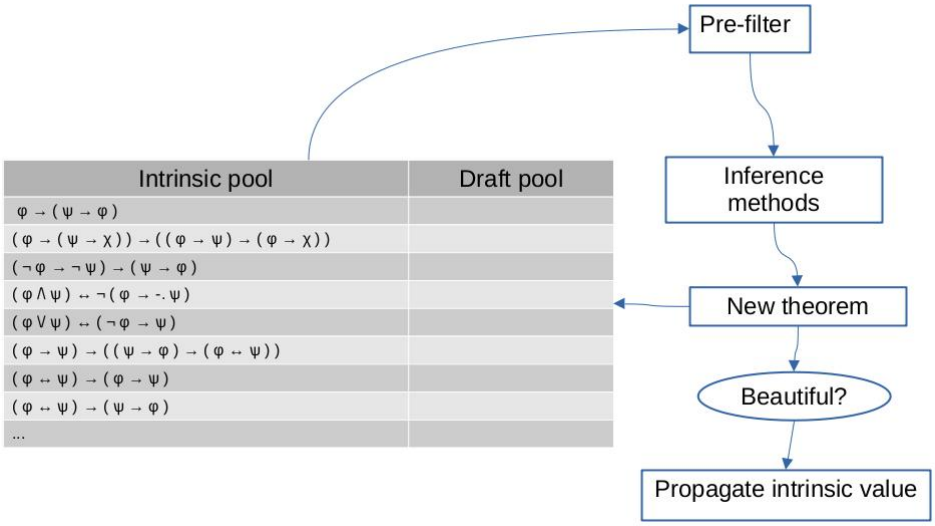
The beauty threshold K then decides how many theorems will be classified as beautiful and indirectly the amount of theorems which will have intrinsic value and serve as training data for the neural networks. Ideally we want a high threshold filtering out all but the most beautiful theorems. Setting it too high will, however, lead to very few intrinsically valuable theorems found. The value is therefore set pragmatically to give enough training data within a reasonable time frame.

4.2 Symbolic system

4.2.1 Overview

The symbolic system is responsible for gathering theorems with intrinsic value, later to be used for training of neural networks. It runs in a Monte-Carlo fashion by randomly deducing new theorems from known ones. As can be seen in the schematic

Figure 4.1: Schematic presentation of the symbolic system



overview in figure 4.1 it consists of two sets of theorems called "pools". One, called the "intrinsic pool" consisting of all theorems with an empirically verified intrinsic value, and another, called the "draft pool" where newly deduced theorems are given a randomly chosen place. The draft pool is limited to be of the same size as the intrinsic pool, causing theorems to be discarded when a newly created theorem takes its place. The theorems in the intrinsic pool, on the other hand, are never overwritten. Instead, the pool grows when new theorems are proven intrinsically valuable. Initially, the intrinsic pool consists of the axioms of the logic in question.

The basic workflow of the symbolic system is as follows:

1. An inference method is chosen randomly
2. The inference method, e.g. modus ponens, uses the pre-filter to gather suitable theorems from both pools, among which it chooses randomly.
3. The inference method uses the randomly chosen theorems to deduce a new theorem.
4. The newly deduced theorem is compared against the existing theorems in both pools, so that it, or any generalization, does not already exist.
5. If the theorem is unique, it is added to the draft pool at a randomly chosen place.
6. A newly added theorem also has its beauty score calculated. If the theorem is considered to be beautiful this beauty score is used to give intrinsic value to all of its parents.
7. Theorems given an intrinsic value are moved to the intrinsic pool, ensuring that they are not overwritten.

4.2.2 Implementation details

The system is written in Python, a choice mostly motivated by our own familiarity with the language and easy integration with the machine learning frameworks. The system simultaneously deals with theorems as flat terms and as graphs of their abstract syntax trees. Since Python is known for its high overhead on function calls, which makes recursion slow, the flat term representation is for quicker unification. Representation as graphs is on the other hand more convenient to work with and is also needed for the GNNs. Each node, or flat term (hereby referred to simply as node), of a theorem, is an object specifying the type of the node, such as operator or variable (for propositional logic), and it's name (for example, "and", "or", "implies", "+", "phi", "0" ...). It also holds a list of all its children. Contrary to Metamath the types of the nodes is hard coded into the system which made implementation easier. The names of the nodes are simply strings to which the underlying system, analogous to Metamath, is mostly agnostic. The exception to this comes as inference rules where the system specifically looks for nodes of certain types as names. For example, the Modus Ponens inference rule is hard coded to look for a theorem with a base node that is an operator with the name "implies". The axioms necessary to define a specific logic are formulated as ASTs, which only differs from any other theorem by a boolean denoting them as axioms, ensuring that they will not be deleted in the continuous updating of the pool contents.

4.2.2.1 Unification

The algorithm we implemented to achieve unification works by simultaneously traversing the tree structure of the two theorems as long as the structure matches. If the logical connectives differs between the theorems the unification obviously fails. If one theorem contains a variable then that variable is unified with the corresponding node, and all it's children, of the other theorem. If the variable to be unified is already unified with something else the unification also fails. For every unification we also checks that no infinite loops arises. For example, setting $\psi = \neg\tau$ and $\tau = \neg\psi$ makes the unification fail.

4.2.2.2 Path indexing

The path indexing module improves efficiency by storing all theorems in a systematic way based on the paths the respective theorems contains. This allows for pre-filtering out any theorem whose structure does not match when searching for generalizations, instances or possible unification candidates for a specific theorem. This greatly improves efficiency compared to looping through all theorems. Our flavor of the classic path index [8] works as follows: Starting from the root of the AST of a theorem and traversing the tree downward to any of its children creates a certain path. From a certain node one can go to the first or second child, indicated by the numbers 1 or 2. This number together with the name of the reached node forms a tuple. A path is encoded as a list of such tuples. For algorithmic consistency, the root node, having no parent, is given the number -1 indicating it occupies no position as a child at any parent. For example, the theorem $\phi \vee \neg\phi$ has the following

paths:

```
(-1, or)
(-1, or), (1, phi)
(-1, or), (2, not)
(-1, or), (2, not), (1, phi)
```

Now, since the names of the variables (phi in the above example), does not matter, we replace them by the type of the variable, such as "wff" or "term", giving the above theorem the following masked paths:

```
(-1, or)
(-1, or), (1, wff)
(-1, or), (2, not)
(-1, or), (2, not), (1, wff)
```

Each such path is used as a key in a python dictionary referring to the specific theorem. The search for generalizations for a theorem is performed by taking all its complete paths, i.e. those ending at a leaf node, and search for generalizations for each such path. A generalization for a path is another path of equal or shorter length but otherwise matching, treating variables as wildcards which matches everything of the same type. For example, theorems which generalizes the path `(-1, or), (2, not), (1, or), (1, wff)` are any theorem containing any of the following paths:

```
(-1, or), (2, not), (1, or), (1, wff)
(-1, or), (2, not), (1, wff)
(-1, or), (2, wff)
(-1, wff)
```

The generalizations for each path are then combined with a set intersection operation leaving only those theorems that has a generalization for every path, i.e. being a (possible) generalization to the whole theorem.
This is sufficient to retrieve possible generalizations of a theorem. However, we also want to be able to search for instances of a path, since paths which belong to possible unification candidates are either generalizations or instances of a specific path. Therefore we employ another dictionary, the *instance index*, where all paths have their endpoint masked to their type. This enables them to be matched to paths ending with a variable. For example, the theorem $\phi \vee \neg\phi$ is indexed to the following paths in the instance index:

```
(-1, wff)
(-1, or), (1, wff)
(-1, or), (2, wff)
(-1, or), (2, not), (1, wff)
```

4.3 Logic

4.3.1 Propositional logic

Initially we explored simple propositional logic following the structure in Metamath [7] library *set.mm*. This logic only deals with the concept of well-formed formulas (WFF). A WFF can be a variable represented by a greek letter, such as ϕ or ψ, which indicates any possible WFF. A WFF can also be an operator, such as $\rightarrow$,$\neg$, $\wedge$, or $\vee$, holding other WFFs as children in an abstract syntax tree (AST). Together with the modus ponens inference rule this completes the underlying structure of this logic. The rest is specified by an arbitrary set of axioms. The ones used for our experiments are listed in table A.1.

4.3.2 Predicate logic with numerical terms and equality

To make things more interesting we extended the simple propositional logic presented above inspired by the specification in Metamath [7] library *peano.mm*. In addition to WFFs, this logic also deals with terms and variables. The implemented logic is quite agnostic toward the actual meaning of terms but in this work they are strictly used to represent some integer value. For example, 0 is a term, r, s, t are terms which analogous to the greek letters can indicate any term. The successor $S(t)$ of a term is also term, so is also $s + t$ and $s * t$. The equality operator, holding two terms as children, for example $s = t$, is a WFF, indicating something that is either true or false. In addition to WFFs and terms there are also "variables", whose meaning differs from the variable WFFs denoted by greek letters. This base type instead denotes a term that can only be used with the implemented quantifiers $\exists$ and $\forall$ (which forms WFFs). Here we deviate slightly from the specification in peano.mm which allows variables to exist outside of quantifiers. This made implementation easier and the space of possible theorems shorter. In peano.mm all axioms are specified for variables but they also hold true for any general term t. The intention for this is presumably that the axioms should be usable within quantifiers. Since our implemented inference methods automatically transform a general term t into a variable x we decided to limit the use of variables to exist only inside quantifiers. In accordance with the specification of the Metamath [7] we also implement the disjoint variable condition, which encodes a requirement for two variables to be distinct.
To this logic we apply five inference methods; modus ponens, substitution, for-all introduction, for-all elimination and exists introduction. Exists-elimination is instead encoded as the axiom (to be used by modus ponens): $\exists x\phi \rightarrow ((\forall x(\phi \rightarrow \psi) \rightarrow \psi)$, where x does not appear in ψ, encoded using the disjoint variable condition.

The rest of the logic is specified as axioms, encoding an augmented version of Robinson arithmetic, which also incorporates divisibility and modulus. Since this logic was intended to mainly focus on mathematics it uses a reduced propositional logic with only implication and negation, specified in table A.5. It also incorporates predicate logic as specified by the axioms in table A.3. The mathematical foundation is specified by the axioms in table A.4.

4.3.3 Inference methods

4.3.3.1 Modus ponens

The modus ponens, stating that $\phi \rightarrow \psi$ and ϕ allows us to infer ψ, is implemented as follows:

1. The path indexing module is queried for all theorems whose top node in its AST is an implication $\rightarrow$.
2. Of all retrieved theorems one is chosen randomly.
3. The chosen implication $\phi \rightarrow \psi$ is divided into its antecedent ϕ and consequent ψ
4. The path indexing module is again queried for theorems whose structure matches ϕ.
5. From the retrieved list of theorems one is chosen randomly and unification with ϕ is attempted. This is continued until success, or until every theorem is tried, in which case the algorithm goes back to step 1, trying another implication.
6. The successful unification is used to derive a new theorem from ψ

4.3.3.2 Substitution

The substitution inference creates new theorems by substituting part of an old theorem with something that is equal. Two WFFs, ϕ and ψ, are considered to be equal if there exists a theorem of the form $\phi \leftrightarrow \psi$. Two terms, s and t, are considered equal if a theorem $s = t$ exists. The process we implemented for performing substitution works as follows:

1. A theorem is chosen randomly from any of the pools.
2. A node in the AST of the theorem is chosen randomly to be substituted
3. If the node is a term, the path indexing module is queried for all theorems of the form $t = s$ and $s = t$, where t is unifiable with the node. If it instead is a WFF (ϕ), the query is for theorems of the form $\phi \leftrightarrow \psi$ and $\psi \leftrightarrow \phi$.
4. From the candidates for substitution, one by one is chosen randomly, and unification is attempted with the WFF or term to be substituted.
5. When unification succeeds the bindings are transferred to the other side of the equality to create a new sub-tree replacing the old node, thereby creating a new theorem.

4.3.3.3 For-all introduction

The for-all introduction inference method is pretty straightforward and works as follows:

1. A theorem is randomly chosen from any of the pools.

2. If there are terms without children (such as s or t) one of them is chosen randomly, otherwise step 1 is repeated.
3. The chosen term is replaced with a new variable and the whole theorem is put inside a for-all node.

4.3.3.4 For-all elimination

This inference rule is a bit peculiar and intended to inject specific numerical terms into the pool of theorems. It works as follows:

1. The path indexing module is queried for all theorems of the form $\forall x \phi$ from which one is chosen randomly.
2. A new term is created by randomly choosing between 0 or a generic term t and appending a random number of successor functions S to it.
3. Every occurrence of x in ϕ is replaced by the newly created term.
4. ϕ is then copied into a new theorem.

4.3.3.5 Exists introduction

The exists introduction we implemented looks for instances of actual numbers in theorems (encoded by an arbitrary number of successor functions), and uses this to create a new theorem stating there exists a number for which this theorem holds. For example, the theorem $S(0) + S(0) = S(S(0))$ can be used to deduce $\exists x\, x + x = S(S(0))$. Specifically it works as follows:

1. A theorem is chosen randomly from any of the pools.
2. From this theorem a term is chosen randomly, among all terms that has no children that are variable terms (thereby only choosing terms indicating actual numbers encoded by an arbitrary number of successor function applications on 0).
3. ϕ is created by replacing all occurrences of the chosen term with a variable x.
4. A new theorem $\exists x \phi$ is created.

4.3.4 Theorem shortening and sorting

Early tests on our symbolic system indicated that it was very prone to creating, as we perceived it, meaningless variations of the same formula. With the introduction of the substitution inference method this became an even bigger issue. Given, for example, the knowledge that $t = t + 0$ the substitution mechanism allows for an exponential number of different ways to express a sum. $1 + 1 = 2$ can be rewritten to $1 + 0 + 1 = 2$, $1 + 0 + 0 + 1 = 2 + 0$ etc. We do not perceive these variations as being beautiful in light of $t = t + 0$ already being known. We deal with this issue similarly to the "Naive term rewriting" algorithm for pruning away redundant formulas, described by Nicholas Smallbone et al. in their paper on QuickSpec [4]. Our

mechanism works by storing every deduced equality, i.e. every discovered theorem of the form $a = b$ or $\phi \leftrightarrow \psi$ into an equivalence class holding all items that are equal. If a newly discovered equivalence belongs to two equivalence classes, those classes are merged into one. Every class is given a representation consisting of one of the shortest formulas of that class. An AST is shortened by traversing the tree, starting with its leaves, and replacing every node belonging to an equivalence class with the representation of that class. This theorem shortening was only performed before a theorem was classified as beautiful or not. The theorems in the pools were not subject to this procedure since we assumed that would remove the possibility of certain deductions.
This shortening of theorems did not remove every possible meaningless variation. For example $1 + 1 = 2$ and $2 = 1 + 1$ are of the same length. While Nicholas Smallbone et al. [4] resolves this issue with "Naive term rewriting" by instead implementing a more sophisticated algorithm called "Knuth-Bendix completion" we have instead chosen a simpler (but not perfect) strategy of sorting terms of unordered operators to slightly remedy the situation. The operators whose children's order does not matter, such as $+$, $*$ $\vee$, $\wedge$, and whose children consists only of operators, had their children kept in lexicographic order. This constraint, to not sort children where any child is a variable, is because variables can be unified with anything whose lexicographic order might be different, thereby limiting the possibility of inferring certain theorems.

4.4 Neural classification

The neural classifiers we have implemented aim to predict the probability, between 0 and 1, of a theorem being intrinsically valuable. The neural classifier thus disregards the actual magnitude of the intrinsic value of a theorem and instead works as a typical binary classifier. The motivation for this type of architecture, compared to estimating the intrinsic value itself, is because it aligns with the way our symbolic system works by answering the question whether a theorem belongs to the intrinsic pool, and can therefore easily be coupled to the decision process of the symbolic system. We also assume that our stochastic exploration, limited by computing resources, of the exponential space of proofs will produce intrinsic values very much subject to randomness. By reducing the classification process to a value between 0 and 1, such "noise" is smoothed out. The theorem is treated as a directed acyclic graph (DAG), where operators and variables are nodes connected by edges. Identical sub-trees in the DAG are merged causing each variable to be represented by a single node. An example of this is shown in figure 4.2. The names of the variables are irrelevant to the neural network. They are all treated similarly on the basis off their type (WFF or term), but are instead represented by individual nodes.

The neural networks used in this work all consist of three parts, the embedding layer, the pooling layer, and the classifier layer. The embedding layer creates an embedding for each node (i.e. operator or variable), whereas the pooling layer transforms the node embeddings into a graph embedding representing a whole theorem. This

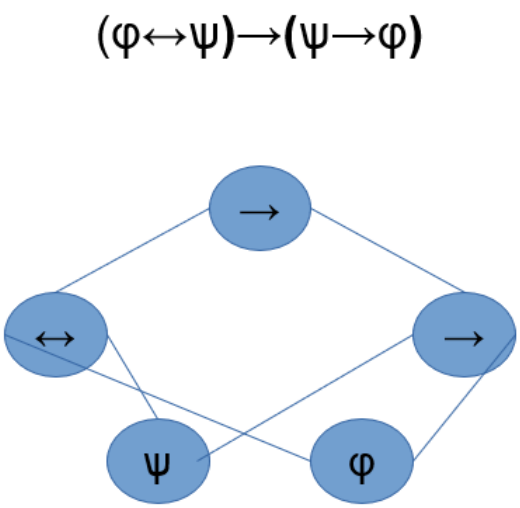


Figure 4.2: Representation of the theorem $(\phi \leftrightarrow \psi) \rightarrow (\psi \rightarrow \phi)$

theorem-representation is transformed by the classifier layer into a value between 0 and 1, representing the probability of a theorem being useful.

Our architectures mostly follow the designs by Maxwell Crouse et al. [6] who in their work deals with two architectures, MPNN and DAG-LSTM. We also experimented with a variation of the MPNN design, referred to as the 'repeating-layer MPNN".

Practically we implement all neural networks in PyTorch and PyTorch Geometric.

4.4.1 Embedding layer

4.4.1.1 MPNN architecture

Initially, each node is given a one-hot encoding (a vector with one element set to 1 and the rest set to 0) depending on which operator or type of variable it represents. The edges are similarly given a one-hot encoding based on the parent node. These one-hot encodings are then fed to linear layers which produce initial node features $h_v^{(0)}$ of size 128 and initial edge features $h_{e_v w}$ of size 64. The node features are then updated through 6 rounds of message passing, a number which seems to produce adequate performance. A round of message passing for layer t consists of aggregating all the parents of a node and all its children into two "messages" $m_{v_p}^{(t)}$ and $m_{v_c}^{(t)}$ according to the following formulas:

$$m_{v_p}^{(t)} = \Sigma_{w \in P_a(v)} F_{M_A}^{(t)}([h_v^{(t-1)} || h_w^{(t-1)} || h_{e_v w}])$$

$$m_{v_c}^{(t)} = \Sigma_{w \in P_c(v)} F_{M_C}^{(t)}([h_v^{(t-1)} || h_w^{(t-1)} || h_{e_v w}])$$

Where $P_a(v)$ and $P_c(v)$ are functions that return the immediate ancestors and chil-

dren of a node. || denotes concatenation. $F_{M_A}^{(t)}$ and $F_{M_C}^{(t)}$ are feed forward networks unique to every layer t with one hidden layer of size 256 and an output layer of size 128, both activated by a tanh function. The two messages are then used, together with the representation from last layer, $h_v^{(t-1)}$ to form a new representation $h_v^{(t)}$ of the node according to the following formula:

$$h_v^{(t)} = h_v^{(t-1)} + F_A^{(t)}([h_v^{(t-1)}||m_{v_p}^{(t)}||m_{v_c}^{(t)}])$$

Where $F_A^{(t)}$ is a feed forward network with a hidden layer of size 384 and an output layer of size 128, each also activated by tanh.

4.4.1.2 Repeating layer MPNN

This architecture is identical to the original MPNN architecture described above with the modification that the weights are shared among the layers, practically implemented as a single layer whose output is fed back as input a number of times.

4.4.1.3 DAG-LSTM

The DAG-LSTM can be seen as a generalization of the Tree-LSTM [11] to DAG-structured data. The main difference being that a Tree-LSTM can batch together node updates on level of depth, whereas the nodes in a DAG require to be grouped together based on a topological sorting. Our implementation of DAG-LSTM is implemented exactly as done by Maxwell Crouse et al. [6], although we will use a feature size of 128 and 64 for nodes and vertices respectively, instead of 256 and 128.
The DAG-LSTM architecture creates embeddings for nodes by aggregating node features stepwise from children to their parents, referred to as an "upward pass", or it can aggregate information from parents to children, called a "downward pass". Another option is to create two sets of embeddings, one from the upward and downward pass respectively, which are then concatenated into one embedding. This is referred to as 'bidirectional DAG-LSTM".

4.4.2 Pooling layer

4.4.2.1 Max pooling

Max pooling is a common pooling technique, available as a layer in PyTorch. It simply creates a graph embedding where every feature is the maximum value of that feature among all the nodes.

4.4.2.2 DAG-LSTM Pooling

By running the DAG-LSTM architecture in an upward pass, it will aggregate information from the bottommost nodes to their parents, and in turn to their parents, and so on, making the topmost node influenced by every node in the graph. In this way the DAG-LSTM architecture can serve as a pooling layer by treating the topmost node as a graph level feature representation.

4.4.3 Classifier layer

The classifier layer consists of a feed-forward neural network with an input layer of size 128 (the feature size), a hidden layer of size 128, and an output layer of size 1. As activation tanh is used for the hidden layer and sigmoid for the output layer.

4.5 Evaluation

4.5.1 Test setup

A basic test run consists of having the symbolic system configured with a set of inference rules and axioms, set to generate a fixed number of theorems. A test run will then find a certain number of beautiful theorems together with the theorems that participated in their deduction (those with intrinsic value). From these test runs, we will also gather theorems without any intrinsic value, to serve as counter-examples in the training of neural networks.

4.5.2 Neural training

The neural networks will be trained, using training data gathered from several test runs, to act as a binary classifier that differentiates between theorems with and without intrinsic value. The theorems will therefore be over-sampled based on their intrinsic value. The training data will be split into training and test sets, and the neural networks will be evaluated based on their results on the test set.

4.5.3 Neurally augmented symbolic system

In order to evaluate the trained neural networks (NNs) we will couple them to the symbolic system and compare the amount of generated beautiful theorems to an identical test run without any neural network. The coupling works by letting the NN give a prediction to each newly generated theorem. We will experiment with having the prediction influence the probabilities of a theorem being chosen for inference. We will also experiment with having the prediction value decide if a theorem is included in the intrinsic pool.

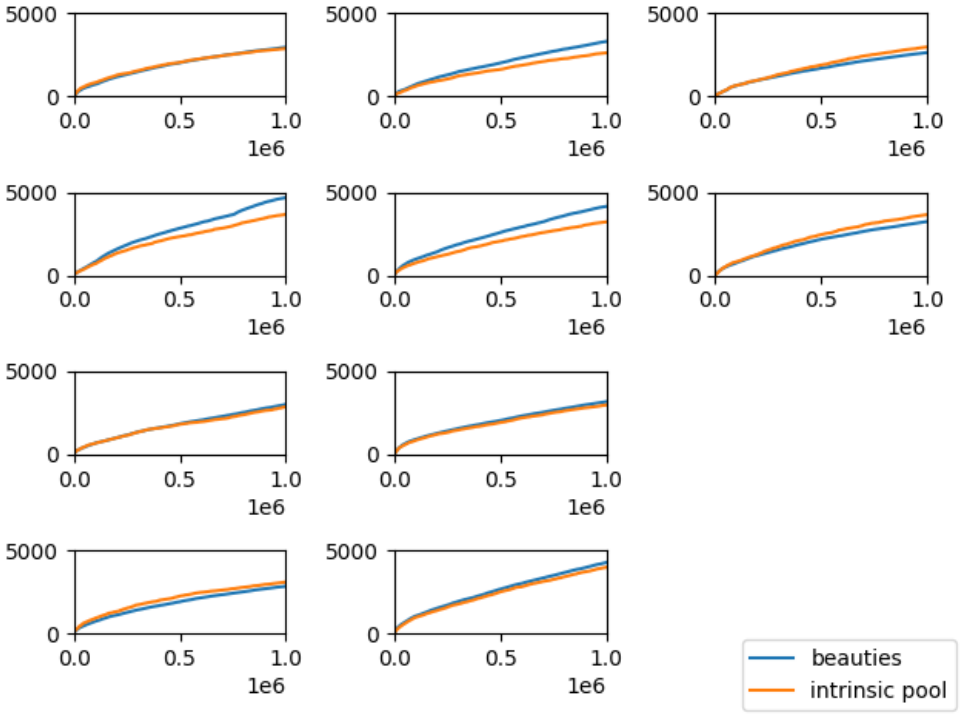
5

Experiments and results

5.1 Theorem generation

To verify the symbolic system and get some insight into its characteristics we performed a test with predicate logic with numerical terms and equality. The test was configured to create 1M theorems, and the beauty limit was set to 0.5. The same test run was executed 10 times and the evolution of found "beauties" and "intrinsics" versus the total amount of created theorems is shown in figure 5.1. As can be seen,

Figure 5.1: Test runs with predicate logic



the number of found beauties and intrinsics were quite influenced by randomness. The number of beauties ranged from 2648 to 4695, averaging at 3436. Similarly the number of intrinsics ranged from 2630 to 3692, with an average of 3203.

5.2 Empirical studies of intrinsic value

5.2.1 Motivation

We wanted to assess how well our devised definition of intrinsic value conforms to reality and support the idea that some theorems are more generally useful than others. In addition, we also wanted to pin point a good value for the alpha constant in our formula for intrinsic value. Recall that the intrinsic value $I(T)$ for a theorem T is defined as $I(T) = \alpha * \Sigma c \in children(T)B(c) + I(c)$ where $B(c)$ denotes the beauty of a "child" (a theorem deduced from the "parent" theorem). The alpha constant then decides how much focus should be put on the long term effects of a theorem. $\alpha = 0$ means we only care about how many beautiful theorems can be directly derived from that theorem (together with another theorem with those inference rules, e.g. modus ponens, that uses two theorems). On the other extreme, $\alpha = 1$ adds to the intrinsic value all beauties that were derived from the theorem directly, from any of its children, grandchildren, and so on.

5.2.2 Test setup

To study this, we first executed several test runs to gather theorems with empirically estimated intrinsic values. The results from these test runs were aggregated together and the intrinsic values were recalculated based on different alpha values. The aggregation was performed by replaying the findings of beautiful theorems from each test-run sequentially (the reason for this strategy, compared to just executing one longer test-run, was due to time and memory limits in the compute cluster). From this aggregation we could select the top N theorems based on a certain alpha-value and compare it to other orderings calculated with a different alpha-value. The different orderings were then evaluated by executing test runs preloaded with the top N (specifically 200, 500, and 2000) theorems from each ordering, and let them produce 1M theorems each. The results, in the form of found beauties, were then compared between the different orderings. To counteract the effects of random variation between results, each test-run was repeated 10 times and the mean values were calculated. For reference, we also performed the same tests with orderings by term length (labeled as "Shortness" in table 5.2), beauty-scores, and a random ordering. We performed this test with both propositional logic and our composition of predicate logic with numerical terms and equality. With propositional logic, we gathered 326K theorems with intrinsic value from a total of 45 test runs set to produce 5M theorems each with a beauty limit set to 0.4.
With predicate logic, we gathered 165K theorems with intrinsic value from a total of 13 test runs configured to produce 10M theorems each. The beauty limit in this case was set slightly higher to 0.5. The number of theorems gathered should not be interpreted to reflect any well thought out strategy, other than reaching sufficient sizes (from our slow Python implementation) before moving on to our next task.

Table 5.1: Avg nr beauties found when preloading with N best theorems from different orderings. (Propositional logic)

Sorting	Beauties (N=200)	Beauties (N=500)	Beauties (N=2000)
$\alpha = 0.001$	5622.8	8361.8	14655.6
$\alpha = 0.01$	5510.2	8360.5	14684.1
$\alpha = 0.05$	4938.0	8489.0	14536.6
$\alpha = 0.1$	4768.7	8620.0	14569.0
$\alpha = 0.15$	4556.1	9309.0	13753.9
$\alpha = 0.2$	5058.5	7593.8	13639.1
$\alpha = 0.3$	5124.1	7298.1	12275.7
$\alpha = 0.4$	2888.6	5103.4	9670.9
$\alpha = 0.5$	2318.2	4457.9	7687.7
$\alpha = 0.7$	2562.5	2380.0	4642.7
$\alpha = 1.0$	1702.5	1786.4	2267.7
Beauty	3085.0	2482.1	5109.0
Shortness	1851.6	2028.0	2518.6
Random	2240.0	2095.6	3921.1

5.2.3 Results

The results of preloading with different orderings of theorems based on propositional logic can be seen in table 5.1 while the results based on predicate logic are shown in table 5.2.
The results show that more beauties were found by preloading with theorems ordered by our calculations of intrinsic value compared to other orderings. This holds for both kinds of logic. Contrary to our assumption, however, the focus on the long term effects of theorems (higher alpha-value) seemed to be only detrimental. With propositional logic, the shortness of a theorem and its related beauty score was better than a random ordering while the contrary was true for predicate logic. To provide possible insight into these results we list the top 20 theorems used for preloading from the ordering of propositional logic based on beauty score in table 5.3, and intrinsic value calculated with $\alpha = 0.001$ in table 5.4. The same is also provided for predicate logic with the top 20 theorems based on beauty listed in table 5.5 and by intrinsic value in table 5.6.
From the negative impact of focusing on long term effects we decided to use a basically non-existent alpha value of 0.001 for the rest of our experiments.

Table 5.2: Avg nr beauties found when preloading with N best theorems from different orderings. (Predicate logic with numerical terms and equality)

Sorting	Beauties (N=200)	Beauties (N=500)	Beauties (N=2000)
$\alpha = 0.001$	2742.4	5816.9	11582.4
$\alpha = 0.01$	2636.5	5831.0	11678.3
$\alpha = 0.05$	2821.8	5572.7	11762.7
$\alpha = 0.1$	2774.1	5323.7	11459.3
$\alpha = 0.15$	2881.3	5605.2	10924.1
$\alpha = 0.2$	2428.6	5046.8	11370.1
$\alpha = 0.3$	2011.3	4335.6	9838.7
$\alpha = 0.4$	1807.3	3516.2	9329.6
$\alpha = 0.5$	1591.8	3454.3	7288.4
$\alpha = 0.7$	1095.0	3126.4	5052.1
$\alpha = 1.0$	970.7	1780.8	3142.0
Beauty score	233.1	144.1	268.2
Shortness	584.4	1151.7	3189.1
Random	962.4	1022.5	1091.0

Table 5.3: Top 20 intrinsics sorted by beauty score. (Propositional logic)

$F \leftrightarrow (\neg T)$
$\phi \rightarrow T$
$T \vee \phi$
T
$(\neg T) \rightarrow F$
$F \rightarrow (\neg T)$
$(\neg T) \rightarrow (F \vee \phi)$
$\phi \rightarrow (F \leftrightarrow (\neg T))$
$(F \rightarrow (\neg T)) \vee \phi$
$\phi \vee (F \rightarrow (\neg T))$
$\phi \vee ((\neg T) \rightarrow F)$
$(\neg F) \rightarrow T$
$(F \leftrightarrow (\neg T)) \vee \phi$
$\phi \vee T$
$F \vee T$
$\phi \rightarrow ((F \leftrightarrow (\neg T)) \wedge \phi)$
$((\phi \wedge F) \vee (\neg T)) \rightarrow F$
$(\neg(F \leftrightarrow (\neg T))) \rightarrow \phi$
$\phi \rightarrow (F \rightarrow (\neg T))$
$(\neg T) \rightarrow (\phi \rightarrow F)$

Table 5.4: Top 20 intrinsics sorted by intrinsic value ($\alpha = 0.001$). (Propositional logic)

$((\neg\phi) \to \chi) \to (\phi \vee \chi)$
$((\phi \wedge \chi) \to (\phi \to \psi)) \to ((\phi \wedge \chi) \to \psi)$
$(\phi \to \chi) \to (\phi \to (\psi \vee \chi))$
$((\phi \to \chi) \to \chi) \to (\chi \leftrightarrow (\phi \to \chi))$
$(\phi \to \chi) \to (\phi \to (\neg(\neg\chi)))$
$((\neg\phi) \to (\neg\psi)) \to (\psi \to \phi)$
$(\phi \to \chi) \to ((\chi \vee \phi) \to \chi)$
$(\phi \to \chi) \to (\phi \to (\chi \vee \psi))$
$((\phi \wedge \chi) \to (\chi \to \psi)) \to ((\phi \wedge \chi) \to \psi)$
$(\phi \vee \chi) \to (\chi \vee \phi)$
$((\neg(\neg\phi)) \to (\phi \to \chi)) \to ((\neg(\neg\phi)) \to \chi)$
$(\phi \vee \chi) \to ((\neg\phi) \to \chi)$
$((\phi \vee \chi) \to \phi) \to (\phi \leftrightarrow (\phi \vee \chi))$
$\phi \to (\phi \vee \psi)$
$(\phi \vee \chi) \to ((\psi \to \chi) \vee \phi)$
$\phi \to (\psi \to \phi)$
$(\phi \to (\chi \vee \psi)) \to (\phi \to (\psi \vee \chi))$
$(\phi \to \chi) \to (\phi \to (\psi \to \chi))$
$(\phi \to (\chi \vee \psi)) \to ((\chi \vee \phi) \to (\chi \vee \psi))$
$(\phi \to (\chi \vee \psi)) \to (\phi \to ((\neg\chi) \to \psi))$

Table 5.5: Top 20 intrinsics sorted by beauty score. (Predicate logic with numerical terms and equality)

$(S0)\|t$
$F \leftrightarrow (\neg T)$
$\neg(0 = (St))$
T
$\forall v T$
$t\|0$
$\phi \rightarrow T$
$\neg(0\|(St))$
$\exists v(F \leftrightarrow (\neg T))$
$\exists v F \leftrightarrow (\neg T)$
$\exists v T$
$F \rightarrow (\neg T)$
$\exists v(\neg(0 = (S(v))))$
$\forall v(\neg(0 = (S(v))))$
$\exists v(\neg(0\|(S(v))))$
$\forall v(\neg(0\|(S(v))))$
$\neg\exists v(0 = (S(v)))$
$\exists v(\neg((v)\|(S0)))$
$\neg\exists v(0\|(S(v)))$
$\neg(((St) + s) = 0)$

Table 5.6: Top 20 intrinsics sorted by intrinsic value ($\alpha = 0.001$). (Predicate logic with numerical terms and equality)

$\exists x\phi \leftrightarrow (\neg\forall x(\neg\phi))$
$((St) = (Ss)) \leftrightarrow (t = s)$
$\exists x\phi \rightarrow (\forall x(\phi \rightarrow \psi) \rightarrow \psi)$
$(s * t) = (t * s)$
$\exists v\phi \rightarrow (\neg\forall v(\neg\phi))$
$\forall v\exists w(\neg(0 = (w)))$
$((St) = (Ss)) \leftrightarrow (s = t)$
$\exists v\forall w(\neg\phi) \rightarrow (\neg\forall v\exists w\phi)$
$(s + t) = (t + s)$
$\exists v(\neg\phi) \leftrightarrow (\neg\forall v\phi)$
$t = ((S0) * t)$
$(t + 0) = t$
$(0 + t) = t$
$t = (0 + t)$
$((S0) * t) = t$
$\forall v(\neg\phi) \leftrightarrow (\neg\exists v\phi)$
$t = (t * (S0))$
$\exists v(\neg\phi) \rightarrow (\neg\forall v\phi)$
$\forall v\exists w(\neg(0 = (S(S(w)))))$
$(t * (S0)) = t$

5.3 Training neural networks to identify intrinsic value

5.3.1 Motivation

The main question of the thesis is whether neural networks can learn to identify "usefulness" or "interestingness", according to our definition of intrinsic value. We also wanted insight into which architectures are more suitable for this task.

5.3.2 Test setup

The intention is to train neural networks as binary classifiers that discriminate between theorems that have intrinsic value and those without. We will investigate three different architectures, an MPNN with 6 different layers, an MPNN with weight sharing across the 6 layers (referred to as "repeating-layer MPNN") and a DAG-LSTM architecture, as described in section 4.4. To train them we will use the same collections of intrinsically valuable theorems used for previous tests, together with 3 times as many theorems, gathered from the same test runs, which showed no intrinsic value, to serve as counter-examples for the training. The factor 3 was chosen slightly arbitrarily in order to utilize the predominance of "useless" theorems while still remaining in the same order of magnitude as the number of valuable theorems. To balance the number of positive versus negative examples we then over-sample the positive ones. We hypothesized that the features of theorems with higher intrinsic value are more important to learn, but we also did not want to waste any gathered training data. Therefore, every theorem T is sampled N times based on its intrinsic value $I(T)$ compared to the mean intrinsic value μ (among the intrinsically valuable ones) according to the formula:

$$N = \lfloor 1 + K * \frac{I(T)}{\mu} \rfloor$$

The value of K was set to 2.5 which was found to create almost as many (96% for propositional logic) positive as negative examples, and a perfect balance was created by discarding some negatives.
When training with theorems from propositional logic we had 326 734 positive examples which were over-sampled into 1 074 256, combined with an equal amount of unique negative examples. The training data based on predicate logic consisted of 165 142 positive examples over-sampled into 552 576.
The training data was divided into a train set (90%) and a test set (10%) and used to train the architectures until convergence.

5.3.3 Results

The results of the predictive performances of the different architectures on the test sets are shown in table 5.7. As shown the prediction accuracy was very similar among the tested architectures, each having an accuracy between 79% - 82%. This was achieved using node and edge feature sizes of 128 and 64 respectively. In other

Table 5.7: Prediction accuracy of different NN architectures

Embedding layer	Pooling Layer	Accuracy (Propositional logic)	Accuracy (Predicate logic)
6 layer MPNN	Max pool	0.8181	0.8165
6 layer MPNN (Repeating-layer)	Max pool	0.8124	0.8113
6 layer MPNN	DAG LSTM	0.8110	0.8164
Bidirectional DAG LSTM	DAG LSTM	0.8063	0.7926

words, we halved the feature sizes used by Crouse et al. [6]. This is because initial tests on the MPNN architecture showed that higher feature sizes only increased accuracy by approximately 1 percentage unit, which we deemed not worth the extra time and computing power needed.

5.4 Neurally augmented theorem generation

5.4.1 Motivation

We have shown that neural networks can, to some degree, learn to discriminate between theorems with intrinsic value and those without. It follows that we also want to put this ability to practical test. Concretely, we want to augment our symbolic system by the neural classifications of theorems and see if this makes it uncover beautiful theorems more efficiently. Given the non-perfect nature of the neural classifications, it is not totally obvious that this would improve anything. One could, for example, speculate that the neural network would misclassify some essential theorems and thereby worsen the results. We also want to assess whether the theorems found with the guiding of the NN are different from those found by a purely stochastic process.

5.4.2 Test setup

We experiment with two strategies. The first is to influence the otherwise random choices of theorems to use for inference by their neural classification. This is achieved by a simple algorithm:

1. Randomly choose one theorem among all possible theorems.
2. Randomly choose a value between 0 and 1. If this value is less than a constant ϵ we have found our theorem.
3. Randomly choose a value between 0 and 1. If the randomly chosen value is higher than the neural classification of the theorem, then start again from step 1, otherwise we have our theorem.

This algorithm influences the test runs in a non-intrusive way which makes test runs comparable to previous test runs. In addition, setting $\epsilon = 1$ provides us with a baseline.
Our second strategy is to let the intrinsic pool be decided directly by a neural network. Since the neural network works as a binary classifier that outputs a value between 0 and 1, indicating the probability that a theorem will be useful, we can do this by adding to the intrinsic pool every inferred theorem with a classification over a certain threshold.
We also test to combine the two strategies such that the neural classification of theorems decides both their likelihood to be used for inference and also whether they are stored in the intrinsic pool.
The neural network used for these tests is the architecture described in section 5.3 with the best performance, namely the 6 layered MPNN embedding layer + Max pooling. The logic used is propositional logic. Each test consisted of generating 1M theorems and was repeated 30 times to counteract the random variability between test runs. The results, in the form of the number of found beauties and intrinsics, were then averaged.

5.4.3 Results

5.4.3.1 Inference selection probabilities influenced by NN classification

Our first test, to influence the probabilities of a theorem being selected for inference, by its neural classification, was performed with e set to 0 (always consider the neural classification), 0.2 (ignore the neural classification 20% of the times) and 1 (never care for the neural classification). The results, presented in table 5.8, unanimously show positive effects of neural classification compared to baseline($\epsilon = 1$).

Table 5.8: Neurally guided choices of theorems for inference

ϵ	Beauties	Intrinsics	Unique beauties	Unique intrinsics
0	9248	8305	5040	6515
0.2	7024	6099	3930	4801
1	2992	2951	1761	2413

5.4.3.2 Intrinsic pool inclusion based on NN classification

The second test, to let inclusion into the intrinsic pool be decided by the neural network, was performed with different values for the inclusion threshold in a quest to pinpoint the optimal value. The results are shown in table 5.9. Since the "intrinsic pool" is not, as previously, populated by theorems that took part in the inference of beautiful theorems, but instead chosen by the neural network, The number of "intrinsics"and "intrinsic pool size" are reported separately. As expected, a lower threshold resulted in a larger size of the intrinsic pool. This increased pool size was

initially accompanied by increased number of found beauties, similar to the results from preloading shown in table 5.1 and 5.2. This increase plateaued at a threshold of around 0.97-0.96, as would be expected by a lower "quality" of the intrinsic pool. In this range, this second strategy performed slightly better than the first.

Table 5.9: Intrinsic pool chosen by neural network

Threshold	Beauties	Intrinsics	Intrinsic pool size	Unique beauties	Unique intrinsics	Unique theorems in pool
0.995	4224	2581	711	1820	1406	400
0.99	7376	4278	2254	3742	2740	1592
0.98	9886	5204	4625	5007	3392	3219
0.97	11503	6305	6459	5590	4057	4106
0.96	10282	6494	7448	5671	4431	4851
0.95	10028	6496	8404	5337	4244	4936
0.94	9177	6528	10460	5166	4433	6742
0.90	4796	4298	14055	2816	3075	10887

5.4.3.3 Combined strategy

Lastly we tested to combine both strategies. The results are shown in table 5.10. Compared to the previous test the combined strategies performed better for every threshold value, otherwise the pattern was similar with an optimal inclusion threshold of 0.97.

Table 5.10: Intrinsic pool chosen by neural network, in combination with neurally guided choices of theorems for inference

Threshold	Beauties	Intrinsics	Intrinsic pool size	Unique beauties	Unique intrinsics	Unique theorems in pool
0.99	10346	6053	2178	4503	3549	1305
0.98	15423	8048	5121	7368	5084	3156
0.97	15904	8680	7279	7660	5575	4177
0.96	12374	7659	9847	6360	5061	6454
0.95	11099	7490	11873	6114	5076	8058
0.90	8458	7483	20545	4953	5380	14997

5.4.3.4 Comparison of beauties found through different configurations

We wanted to see if there were differences between sets of beauties found by the various methods, to what extent they stumbled upon the same beautiful theorems. The sets chosen were the baseline (no aid by NN), labeled "B", together with the sets of the combined strategy presented in table 5.10, labeled by their threshold value, denoted by "T". To produce a fair comparison given the different numbers of

found beauties we chose a subset consisting of the 45 803 theorems (The size of the smallest set) from each set. These subsets were formed by looping through the test runs one by one and adding every theorem until the subset was of size 45 803. For reference we wanted to compare configurations to themselves. To do this we created extra subsets from disjoint sets of test runs. The ratios of the number of theorems common among two different subsets are shown in table 5.11.

Table 5.11: Similarity ratios between sets of beauties found by different configurations

	B	T=0.90	T=0.95	T=0.96	T=0.97	T=0.98	T=0.99
B	30%	22%	22%	24%	18%	21%	17%
T=0.90	22%	23%	22%	22%	22%	20%	16%
T=0.95	22%	22%	28%	25%	20%	23%	20%
T=0.96	24%	22%	25%	29%	23%	24%	16%
T=0.97	18%	22%	20%	23%	23%	23%	15%
T=0.98	21%	20%	23%	24%	23%	26%	20%
T=0.99	17%	16%	20%	16%	15%	20%	33%

The average similarities were quite consistent. However, slight difference can be seen when comparing the similarities of identical configurations, which average 27%, to different configurations with an average similarity of 22%. This is especially apparent for T=0.99 where the similarity to an identical configuration is 33% while the average similarity compared to other configurations is 17%.
To dig deeper into the issue we performed another comparison. This time by comparing each of our subsets, of size 45 803, to the 216 496 beauties found when gathering training data (i.e. no neural augmentation). The results are shown in table 5.12.

Table 5.12: Similarity ratios between beauties from training data vs different configurations

B	T=0.90	T=0.95	T=0.96	T=0.97	T=0.98	T=0.99
30.6%	25.3%	23.1%	23.4%	20.2%	22.5%	17.7%

Surprisingly, this comparison differed only slightly from the comparison to the baseline, even though this dataset was almost 5 times larger. This suggests that some theorems are “low hanging fruit”, easier to find than others. Since each of our sets were comprised of 30 individual test runs we could count how many times each theorem was discovered. Then we could create lists of the 1000 most common theorems from each set, and see how many of them were present in the other sets. This comparison is shown in table 5.13.

From this table it is clear that beauties that were easy to find by one configuration is also easily found by others. It is also clear that similar configurations tend to produce similar theorems, which is also in line with the results from table 5.11. The same pattern is also visible when comparing the top 1000 selections to each other in table 5.14.

Table 5.13: Presence of the 1000 most easily found theorems in other sets

Top 1000 \Whole set	B	T=0.90	T=0.95	T=0.96	T=0.97	T=0.98	T=0.99
B		99.9%	99.9%	99.6%	99.3%	98.3%	97.6%
T=0.90	99.1%		100.0%	100.0%	99.9%	99.9%	98.3%
T=0.95	99.4%	100.0%		100.0%	100.0%	99.8%	99.1%
T=0.96	97.1%	100.0%	100.0%		100.0%	100.0%	99.7%
T=0.97	94.2%	100.0%	100.0%	100.0%		100.0%	99.8%
T=0.98	97.8%	99.8%	99.9%	100.0%	100.0%		99.9%
T=0.99	91.8%	98.0%	99.3%	99.7%	100.0%	100.0%	

Table 5.14: Similarities between top 1000 most easily found theorems by different configurations

	B	T=0.90	T=0.95	T=0.96	T=0.97	T=0.98	T=0.99
B		60.7%	63.5%	56.5%	49.6%	50.0%	46.6%
T=0.90	60.7%		68.3%	67.1%	60.9%	57.3%	46.7%
T=0.95	63.5%	68.3%		74.1%	65.6%	59.7%	52.6%
T=0.96	56.5%	67.1%	74.1%		74.5%	64.0%	57.5%
T=0.97	49.6%	60.9%	65.6%	74.5%		61.2%	60.0%
T=0.98	50.0%	57.3%	59.7%	64.0%	61.2%		57.4%
T=0.99	46.6%	46.7%	52.6%	57.5%	60.0%	57.4%	

We also looked at the average beauty scores found using the different configurations. As seen in table 5.15 there are no major differences.

Table 5.15: Average beauty scores of beauties found through different configurations

B	T=0.90	T=0.95	T=0.96	T=0.97	T=0.98	T=0.99
0.437	0.430	0.428	0.428	0.426	0.428	0.426

6

Conclusion

6.1 Conclusion

We have demonstrated, in section 5.2 that some theorems are inherently more useful than others. We expected, however, that some theorems would confer some sort of breakthrough similar to e.g. how Euler's identity have enabled the deduction of many new theorems. This is what we were trying to model with the parameter alpha which would increase the intrinsic value of a theorem based on deductions further down the line, from it's children, grandchildren, and so on. However, no such kind of central significance of any theorem was ever detected. Instead, the usefulness only seems to relate to a theorem's ability to directly infer beautiful theorems. From table 5.1 we see that the alpha value was purely detrimental. We suspect that the signal from the alpha value might have been noisy due to too few theorems. Another source of random noise comes from our simulation only allowing a theorem to have one set of parents, even though it might be inferred through a multitude of pathways.

In section 5.3 we have shown that neural networks can indeed learn to identify usable theorems. This was best done with a simple MPNN followed by max pooling. This MPNN could also share weights between layers without notable loss. Interestingly, the bidirectional DAG-LSTM with DAG-LSTM pooling did worse than the MPNN, which is the opposite of the results by Crouse et al. [6] which utilized NNs to select premises for theorem proving. The differences in performance between the architectures were, however, not significant.
We also showed, in section 5.4, that relying on the assessment of a trained neural network, to decide suitable candidates for inference, can substantially improve the outcome of our simulation by making the discovery of beautiful theorems much more probable. Our tests show improvements up to a factor 4.
From table 5.13 it is evident that some theorems are more likely to be found than others. It is also evident that most of the theorems easily found by a pure stochastic simulation are also easily found when relying on an NN for guidance, and vice versa. The NN does, however, introduce a slight shift in the probability distribution. This is especially apparent when the NN is coupled with a high threshold for selecting inference candidates. Although we have no evidence thereof we can not rule out the possibility that relying too much on the NN predictions would make the discovery of certain theorems impossible.

6.2 Discussion

The stochastic method of theorem generation used in this project has a natural tendency to produce countless variations of the same kind of theorem. Creating a definition of beauty that is rigorous enough has presented us with great difficulties. Our efforts on this issue, e.g. implementing equality classes and reducing a theorem to its shortest possible representation, have greatly improved the matter. However, the system still floods us with countless theorems of the form $1+1=3 \rightarrow 2+2=5$, which, although logically correct, feels far from being beautiful. For the system to avoid classifying the above theorem as beautiful, it must first have deduced $\neg(1+1=3)$, which in turn constitutes another meaningless formula among all the other stuff that is not equal to 3. It is questionable whether a theorem that participates in the creation of something meaningless can be considered valuable. However, this issue is partly due to the reduction mechanism not acting retroactively. If that were the case, $\neg(1+1=3)$ would be discarded as soon as $(St+St=SSs) \leftrightarrow (t+t=s)$ which would reduce the Peano representation of $1+1=3$ to $0+0=1$ which would be reduced to $0=1$.

6.3 Future work

Here follows some ideas for future work, would anyone be interested in continuing on this path:

- The symbolic system would benefit greatly from moving away from its Python implementation into a faster language.
- Path filtering should be discarded in exchange for the complicated but much more efficient "Substitution tree indexing" [16]. This pre-filtering technique offers lookup for generalizations, instances and unifications, with a time complexity of $O(1)$.
- Employ the reduction mechanism on beautiful formulas retroactively, thereby discarding those that can no longer be considered beautiful in light of new knowledge.
- There are probably better choices of neural architecture for this task. This has not been thoroughly investigated. Perhaps GAT would be a good choice. A theorem's usability also often depends on other theorems. Perhaps a neural network should aggregate information from all known theorems instead of its own structure alone. Only focusing on the structure of a theorem requires the neural network to contain mathematical "understanding" in its weights. A neural network that gets its knowledge externally becomes more adaptive, thus being a better theory explorer.
- Include the ability to disqualify other theorems from being beautiful in the definition of intrinsic value, since it is actual proof of usability.

Bibliography

[1] H. Xin et al., "Deepseek-prover-v1. 5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search," *arXiv preprint arXiv:2408.08152*, 2024.

[2] G. Poesia, D. Broman, N. Haber, and N. Goodman, "Learning formal mathematics from intrinsic motivation," *Advances in Neural Information Processing Systems*, vol. 37, pp. 43 032–43 057, 2024.

[3] Y. Bengio and N. Malkin, "Machine learning and information theory concepts towards an ai mathematician," *Bulletin of the American Mathematical Society*, vol. 61, no. 3, pp. 457–469, 2024.

[4] N. Smallbone, M. Johansson, K. Claessen, and M. Algehed, "Quick specifications for the busy programmer," *Journal of Functional Programming*, vol. 27, e18, 2017.

[5] S. H. Einarsdóttir, N. Smallbone, and M. Johansson, "Template-based theory exploration: Discovering properties of functional programs by testing," in *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, 2020, pp. 67–78.

[6] M. Crouse et al., "Improving graph neural network representations of logical formulae with subgraph pooling," *arXiv preprint arXiv:1911.06904*, 2019.

[7] N. Megill and D. A. Wheeler, *Metamath: a computer language for mathematical proofs*. Lulu. com, 2019. [Online]. Available: `http://us.metamath.org/downloads/metamath.pdf`.

[8] M. E. Stickel, "The path-indexing method for indexing terms," Tech. Rep., 1989.

[9] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[10] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[11] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[12] T. Skolem, "Peano's axioms and models of arithmetic," in *Studies in Logic and the Foundations of Mathematics*, vol. 16, Elsevier, 1955, pp. 1–14.

[13] D. P. Kroese and R. Y. Rubinstein, "Monte carlo methods," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 4, no. 1, pp. 48–58, 2012.

[14] D. Wells, "Are these the most beautiful?" *The Mathematical Intelligencer*, vol. 12, no. 3, pp. 37–41, 1990.

[15] S. Zeki, J. P. Romaya, D. M. Benincasa, and M. F. Atiyah, "The experience of mathematical beauty and its neural correlates," *Frontiers in human neuroscience*, vol. 8, p. 68, 2014.

[16] P. Graf, "Substitution tree indexing," in *International Conference on Rewriting Techniques and Applications*, Springer, 1995, pp. 117–131.

A

Appendix 1

A.0.1 Axioms

Table A.1: Axioms used for propositional logic

$\phi \to (\psi \to \phi)$
$(\phi \to (\psi \to \chi)) \to ((\phi \to \psi) \to (\phi \to \chi))$
$(\neg\phi \to \neg\psi) \to (\psi \to \phi)$
$(\neg\phi \to \psi) \to ((\neg\phi \to \neg\psi) \to \phi)$
$\phi \to (\psi \to (\phi \wedge \psi))$
$(\phi \wedge \psi) \to \phi$
$(\phi \wedge \psi) \to \psi$
$\phi \to (\phi \vee \psi)$
$\phi \to (\psi \vee \phi)$
$(\phi \to \chi) \to ((\psi \to \chi) \to ((\phi \vee \psi) \to \chi))$
$(\phi \leftrightarrow \psi) \to (\phi \to \psi)$
$(\phi \leftrightarrow \psi) \to (\psi \to \phi)$
$(\phi \to \psi) \to ((\psi \to \phi) \to (\phi \leftrightarrow \psi))$
$((\phi \wedge \psi) \leftrightarrow \neg(\phi \to \neg\psi))$
$((\phi \vee \psi) \leftrightarrow (\neg\phi \to \psi))$
$(\phi \to \phi) \leftrightarrow T$
$F \leftrightarrow \neg T$

Table A.2: Axioms of Robinson arithmetic

$\neg(S(t) = 0)$
$(S(t) = S(s)) \to (t = s)$
$(t = 0) \vee \exists x(S(x) = t)$ (x, t disjoint)
$t + 0 = t$
$t + S(s) = S(t + s)$
$t * 0 = 0$
$t * S(s) = t * s + s$

Table A.3: Axioms for predicate logic

$(\exists x\phi) \leftrightarrow \neg(\forall x\neg\phi)$
$(\forall x\phi) \leftrightarrow \phi$ (x and ϕ disjoint)
$(\exists x\phi) \leftrightarrow \phi$ (x and ϕ disjoint)
$(\exists x\phi) \rightarrow ((\forall x(\phi \rightarrow \psi) \rightarrow \psi)$ (x and ϕ disjoint)
$(\forall x(\phi \rightarrow \psi) \rightarrow ((\forall x\phi) \rightarrow (\forall x\psi))$

Table A.4: Axioms for extended Robinson arithmetic

$\neg(S(t) = 0)$
$(S(t) = S(s)) \leftrightarrow (t = s)$
$\neg(t = 0) \rightarrow \exists x(S(x) = t)$ (x, t disjoint)
$t + 0 = t$
$t + S(s) = S(t + s)$
$t * 0 = 0$
$t * S(s) = t * s + s$ $t + t = S(S(0)) * t$
$t = t$
$T \leftrightarrow t = t$
$s + t = t + s$
$s * t = t * s$
$r * (s + t) = r * t + s * t$
$r * (s * t) = (r * s) * t$
$r + (s + t) = (r + s) + t$
$s * t = 0 \leftrightarrow ((\neg s = 0) \rightarrow t = 0)$
$s * t = S(0) \rightarrow s = S(0)$
$t = S(0) * t$
$s\|t \leftrightarrow \exists x s * x = t$ (x not in s or t)
$s\|t \leftrightarrow t\%s = 0$
$(s * t)\%t = 0$
$(r + s * t)\%t = r\%t$

Table A.5: Reduced set of propositional logic axioms

$(\phi \rightarrow (\psi \rightarrow \phi))$
$((\phi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\phi \rightarrow \psi) \rightarrow (\phi \rightarrow \chi)))$
$((\neg\phi \rightarrow \neg\psi) \rightarrow (\psi \rightarrow \phi))$
$((\phi \rightarrow \psi) \rightarrow ((\psi \rightarrow \phi) \rightarrow (\phi \leftrightarrow \psi)))$
$((\phi \leftrightarrow ps) \rightarrow (\phi \rightarrow \psi))$
$((\phi \leftrightarrow ps) \rightarrow (\psi \rightarrow \phi))$
$(\neg(\neg\phi)) \leftrightarrow \phi$
$(\neg\phi \rightarrow \psi) \rightarrow ((\neg\phi \rightarrow \neg\psi) \rightarrow \phi)$
$(\phi \rightarrow \phi) \leftrightarrow T$
$F \leftrightarrow \neg T$