Department of Computer Science
Work Group: Artifical Intelligence

MIXING AUTOMATED THEOREM PROVING AND MACHINE LEARNING

Bridging the gap between numerical and symbolic reasoning with a potpourri of computer science

submitted by
MARCO ZIENER
MATR.-NR.: 4359973
m.ziener@fu-berlin.de

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Date of submission:	November 19, 2017
Primary Examiner:	Priv.-Doz. Dr.-Ing. Christoph Benzmüller
Second Examiner:	Prof. Dr. Raúl Rojas
Supervision:	Max Wisnewski
	& Alexander Steen

Marco Ziener: *Mixing automated theorem proving and machine learning ,*
Bridging the gap between numerical and symbolic reasoning with a
potpourri of computer science , © November 19, 2017

From a Curry-Howard perspective,
"I like programming but I don’t like maths."
means
"I like programming but I don’t like programming.".

— Conor McBride

ABSTRACT

Artificial Intelligence suffers a dichotomy between symbolic and numeric methods. With the advent of Deep Learning and the good performance on several different fields, its applications in context of the automated theorem prover Leo-III are explored. By applying methods from functional programming, this thesis introduces a reconstruction routine for proofs which allows to execute them and subject them for further analysis. Furthermore, the role of clause selection in internal proof-guidance is explored by modeling the proof process as a Markov Decision Process without clause selection and the application of a Markov Decision Process for optimizing the clause selection is described. In the end, further applications of machine learning techniques in Leo-III are outlined.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [44]

ACKNOWLEDGEMENTS

Without the countless people whose ideas I could cannibalize this thesis would not have been possible.

Professionally, I would like to thank Christoph for rekindling my passion for computer science with his lectures on logic and Alex and Max for having the endurance keeping up with my questions.

On a personal level, I would like to thank Andrea for making the gruesome time of writing as pleasant as possible and my mother for her everlasting support.

CONTENTS

LIST OF FIGURES

LISTINGS

ACRONYMS

ATP Automated theorem proving/prover

BLISTR Blind Strategymaker

CADE Conference on automated deduction

CASC CADE ATP System Competition

JEP Java Embedded Python

MDP Markov Decision Process

MALES Machinea Learning Strategies

RELU Rectified Linear Unit

SETHEO Sequential theorem prover

STT Simple Type Theory

TPTP Ten-thousands Problems for Theorem Provers

1 INTRODUCTION

Since the Logic Theorist in 1955 [67], automated theorem proving has gained significant traction. During the last few years standard protocols for communicating problems and results [84, 85], the increasing computational power and memory present in current computer systems [78] and improvements in the theoretical calculi[1] allowed applications in various domains like hardware verification [23], metaphysics [9] and mathematics [59]. Automated theorem provers aspire to emulate the human proof process by rephrasing the human proof process as a search problem. Although improvements were made, the search process done by the prover remains rather primitive and does not even resemble closely how a human would approach these problems: Meanwhile, a human being is guided by intuition, experience and understanding during the proving process, an automated theorem prover is not. It infers *knowledge* from the initial premises until it finds the desired solution. As a consequence, the human being performs better even in large proofs because it can recognize and select what approach to take and which parts of the premises are more important than others.

For automated theorem provers the problem is the expansive search space and making sensible decisions on which axiom to expand and which inference rule to use for the application on a term. Almost all systems try to rectify this problem by interleaving several modes which in turn resemble certain human heuristics. Although, this approach has been proven effective on *small* proofs, it deteriorates quickly when the search spaces become large. Still, in order to derive the necessary proof often only a fraction of the generated information is actually necessary but an immense amount of superfluous inferences are made.

The strategies to strengthen the performance of theorem provers on large scale search spaces involve user interaction for guidance i.e., Coq [7] and Isabelle [69], the usage of highly optimized data structures and term rewriting to avoid computational expensive operations if possible. Albeit these approaches are important they can only mitigate the underlying issue: The automated theorem prover does not know what it is doing and just applies inferences rules to the knowledge. Of course, this approach has the large advantage that it scales well with the ever-going improvements to hardware. Yet an interesting example is chess where – even though the speed of com-

1 For instance cut elimination where McDowell and Miller [60] present one of many results.

puters increased massively – the role of a massive amount of judging positional factors, aggressive pruning of the search tree and the successful combination in evaluating a position is the difference between the top engines. Even stronger improvements have been on the game of Go [80, 81]. By an approach made of deep learning and randomized search, super human performance has been achieved much faster than anticipated by the artificial intelligence community.

For automated theorem proving, such a diverse and well-formulated framework of factors in the human proof process does not exist yet as it does for Go or chess. The formulation of useful heuristics is rather difficult – even for human beings – since each problem has unique characteristics and involves a certain amount human creativity and insight. Additionally, the foundation and presentation of proofs is another highly discussed topic. Questions like *What is the right foundation of mathematics?* and *How should proofs be presented?* provide much discussion among the mathematical community. For automated theorem provers the question is less of a philosophical nature but more of a pragmatic one albeit the results naturally raise the above questions[2].

Recently it was shown that although using machine learning techniques as part of the proving process sacrifices raw computational resources, it is an investment that has a high pay-off [27]. This is a promising result as the employed techniques are relatively simple.

In the recent years the deep learning approach has become very popular over many fields of artificial intelligence yet largely omitted the knowledge representation and automated reasoning. Especially in the later, it is not unreasonable to give up some raw computation power to make smarter decisions concerning the traversal of the search space. But for that to happen the already described problem of recognizing what is important in a proof process must be addressed before constructing the appropriate model for learning process and training.

2 For instance, regard the long process of proving Kepler's conjecture by Hales [37] and the enormous size (13 GB) of Erdös discrepancy machine proof [46]

2

PRELIMINARIES

This chapter introduces the general building blocks of this thesis. Note that it is a mere introduction to these topics and not an exhaustive treatment and seeks to supply the reader with the necessary terminology to understand the work which has been done in Chapter 3 as well as provide a reference point for further studies in the case of interest. The outline and description of these topics in the thesis, loosely follows the mentioned literature.

For an introduction to higher-order logic based on theory of simple types, consider the Stanford Encyclopedia of Philosophy [3] or the article by Farmer [29]. As a reference, the seminal book is by Andrews [5]. On the aspects of intensionality and extensionality the work of Muskens [66] and Benzmüller, Brown and Kohlhase [12] provides solid ground to start. Concerning the embedding of other logics into the simple theory of types the starting point is Benzmüller and Paulson [11] with a plethora subsequent publications culminating in the analysis of Gödels ontological argument [10].

Category theory itself has a very large body of theory surrounding it with a rather steep learning curve and countless applications. The quick and easy introduction without much mathematical rigor but with focus on the applications is Spivak [82]. Of course, the seminal work is by MacLane [55] but it requires a lot of mathematical maturity. Applications to logic can be found in Jacobs [40] and in Lambek [50].

Artificial intelligence has undergone a huge and unfair reordering up to the point where it has become synonymous with machine learning and deep learning. Several sub-fields have shifted nearly completely out of focus like logic and probabilistic reasoning, knowledge representation and evolutionary programming. However, a general and broad introduction to artificial intelligence is still the book by Norvig and Russel [77]. An in-depth overview of neural networks is given by the book of Rojas [74]. A mathematically compact notation which is closer to the hardware – and on which the presentation here is based, since neural networks are only applied as black-boxes – provides Nielsen [68]. For deep learning the recent book by the founding fathers of that field Goodfellow, Bengio and Courville [36] provides a theoretical introduction as well as a reference point. The practical aspects are explored in Gibson [35], Géron [34] and Buduma [17].

Concerning reinforcement learning, Sutton and Barto [88] have written the seminal textbook which provides an overview on the subject. Because of the age it does not contain the newest developments in the field. In particular, these new developments use machine learn-

ing techniques in order to make the learning process more efficient and expand the possible field of application. For this purpose neural networks of different size and shape have been used. An overview of these developments, as well as a general introduction, is found in the Berkeley deep reinforcement boot-camp [1] as well as the course by David Silver [2]. There is also a rewrite of the Sutton and Barto's book in work which should address these developments eventually.

Furthermore, some knowledge of linear algebra, analysis and probability theory will be assumed.

2.1 CHURCH'S SIMPLE THEORY OF TYPES

The simple theory of types (**STT**) is a formulation of higher-order logic which is well suited for mechanization and automated theorem proving. It is based on Church's work combining type-systems for math with the computational model of the λ-calculus in 1940 [19]. Since then, it has been extensively studied by mostly his students – Andrews [5, 4] and Henkin [38] – and became theoretical very well-founded and explored. Furthermore, due to the theoretical founding and the inherent expressiveness it is relativly simple to embed other logics into **STT** while preserving their semantics [11]. Together with the low amount of inference rules and their relative simple nature it excels in practical application and provides very useful abstractions for problem-solving.

As the name implies the foundation of **STT** is the notion of a type. It is the starting point for a solution to the infamous Russell-Paradox [76] which pointed out the formal problems in Frege's formulation of quantification and sets [31]. Eventually, this lead to Russell and Whitehead [99] formulating a new foundation for mathematics by introducing types. Intuitively types can be seen as *tags* for the expressions of a language which add additional information and restrictions to expressions. Since the expressions must obey the rules of the type system to be valid expressions of the language, this makes problematic expressions unrepresentable since they can not be assigned a valid type.

Church[1] constructed **STT** by combining the computational λ-calculus with a type system with the intention of expressing a logic.

The construction begins by defining a base type σ which represents the truth values and – since the logic can be easily extended – another type ι which allows the user to expand and adapt the logic to his individual needs. In contrast to other logical systems, simple type theory has a built-in way to generate new types based on already defined ones. Formally:

1 Historically in the 1920s and 1930s, several mathematicians tried to further refine, simplify and fix problems in the *Principia*. An incomplete but prominent list includes Carnap [18], Chwistek [21] and Ramsay [72].

Definition 2.1.1 (**Base Types,** σ**,** ι**,** $\rightarrow$):
The types of simple type theory are generated from the following base types:

1. *Let* σ *be the type of Boolean values with two distinct logical constants:* $\top$ *for true and* $\bot$ *for false.*
2. *Let* ι *be the type of individual values.*
3. *Let* α *and* β *be arbitrary types, then the type* $\alpha \rightarrow \beta$ *denotes the functions which map terms of type* α *to terms of type* β*.*

The set of types $\mathcal{T}$ *is freely generated by the application of the right associative function application constructor* $\rightarrow$ *on these two base types.*

Note that the letters of the Greek alphabet but σ and ι are used to indicate an arbitrary type and that the types will be omitted if they can be reconstructed from the context.

Definition 2.1.2 (**Signature**):
Let $\mathcal{V}_\alpha$ *denote a denumerable and infinite set of variable symbols for type* α*, let* $\mathcal{T}$ *be the set of all types and let* $\mathcal{C}_\alpha$ *be the set of all constants of a type* α*. Then let* $\Sigma = (\mathcal{V}, \mathcal{C})$ *be a signature such that* $\mathcal{V} = \bigcup_{\alpha \in \mathcal{T}} \mathcal{V}_\alpha$ *and* $\mathcal{C} = \bigcup_{\alpha \in \mathcal{T}} \mathcal{C}_\alpha$*.*

Depending on the intention of the user, there are many ways to derive the language of **STT**. An option is to use equality as a primitive and derive all the other operations as abbreviations from it. This path is taken by Andrews [5] but for mechanization, it is simpler to use another option: Derive the equality through primitive logical operations which form a functional basis and combine them with principles of extensionality. Both of the approaches are equal in terms of expressivity but checking equality is harder in mechanization due to requiring a complete traversal of the term structure. Since this work is concerned with logic in the context of automated theorem provers, the second option is the natural choice.

Definition 2.1.3 (**Syntax of typed** λ**-calculus**):
The abstract syntax of ***STT*** *can be given in a short Backus-Naur-Form-like grammar with* ν_α *being variables of type* α *and* ρ_α *being a constant of type* α*:*

$$\begin{aligned} s, t ::= {} & \rho_\alpha \mid \nu_\alpha \mid (\lambda \nu_\alpha . s_\beta)_{\alpha \rightarrow \beta} \mid (s_{(\alpha \rightarrow \beta)} t_\alpha)_\beta \mid (\neg_{\sigma \rightarrow \sigma} s_\sigma)_\sigma \\ & \mid (\vee_{\sigma \rightarrow \sigma \rightarrow \sigma} s_\sigma t_\sigma) \mid (\Pi_{(\alpha \rightarrow \sigma) \rightarrow \sigma} s_{\alpha \rightarrow \sigma})_\sigma \end{aligned}$$

The primitives of choice are the Boolean connectives $\vee$ *and* $\neg$ *together with the logical constant* $\Pi_{(\alpha \rightarrow \sigma) \rightarrow \sigma}$*. The rest of the connectives are defined in the usual way as abbreviations.*

The quantification $\forall \nu_\alpha . s_\sigma$ in **STT** is an abbreviation which is defined in terms of the more general logical constant: $(\Pi_{(\alpha \rightarrow \sigma) \rightarrow \sigma} (\lambda \nu_\alpha . s_\sigma))$. This constant is assumed to exist for each type and if given $(\lambda \nu_\alpha . t_\sigma)$ it evaluates it on all the inhabitants of that type α and returns true if it

is true on all of them. Otherwise it returns false. Due to the existence of Π for every type α this constant allows to quantify over arbitrary sets.

To further enrich the language, a few syntactical operations have to be introduced which connect the various syntactical structures with each other. These operations are based on the bound and free occurance of a variable.

Definition 2.1.4 (Free, Bound):
The set of free variables of a term T *is denoted* ***Free***(T) *and is calculated inductivly:*

$$\begin{aligned}
&\mathbf{Free}(x_\alpha) = \{x_\alpha\}\\
&\mathbf{Free}(SR) = \mathbf{Free}(S) \cup \mathbf{Free}(R) \textit{ for terms } S \textit{ and } R\\
&\mathbf{Free}(\lambda x_\alpha.S) = \mathbf{Free}(S) \setminus \{x_\alpha\}
\end{aligned}$$

In similar fashion the bound variables can be calculuated:

$$\begin{aligned}
&\mathbf{Bound}(x_\alpha) = \emptyset\\
&\mathbf{Bound}(SR) = \mathbf{Bound}(S) \cup \mathbf{Bound}(R) \textit{ for terms } S \textit{ and } R\\
&\mathbf{Bound}(\lambda x_\alpha.S) = \mathbf{Bound}(S) \cup \{x_\alpha\}
\end{aligned}$$

With the free and bound variables dealt with, it is now possible to define substitution on **STT**.

Definition 2.1.5 (Substitution):
A substitution is a mapping which allows replacing free variables in an expression by another term if certain criteria match. A substitution is written

$$T_\gamma[x_\alpha \leftarrow S_\alpha]$$

which means that in the Term T_γ *each occurrence of* x_α *is replaced by* S_α *regarding the following rules:*

1. $x_\alpha[x_\alpha \leftarrow S_\alpha] = S_\alpha$
2. $y_\beta[x_\alpha \leftarrow S_\alpha] = y_\beta$ *if* $y_\beta \neq x_\alpha \wedge (y_\beta \in \mathcal{V}_\beta \vee y_\beta \in \mathcal{C}_\beta)$
3. $(T_{\beta\to\alpha}R_\beta)[x_\alpha \leftarrow S_\alpha] = ((T_{\beta\to\alpha}[x_\alpha \leftarrow S_\alpha])(R_\beta[x_\alpha \leftarrow S_\alpha]))$
4. $(\lambda x_\alpha.T_\beta)[x_\alpha \leftarrow S_\alpha] = (\lambda x_\alpha.T_\beta)$
5. $(\lambda y_\beta.T_\gamma)[x_\alpha \leftarrow S_\alpha] = (\lambda y_\beta.T_\gamma[x_\alpha \leftarrow S_\alpha])$ *if* $x_\alpha \neq y_\beta \wedge (y_\beta \notin \mathbf{Free}(S_\alpha))$
6. $(\lambda y_\beta.T_\gamma)[x_\alpha \leftarrow S_\alpha] = (\lambda z_\beta.T_\gamma[y_\beta \leftarrow z_\beta][x_\alpha \leftarrow S_\alpha])$ *if* $y_\beta \in \mathbf{Free}(S_\alpha) \wedge x_\alpha \in \mathbf{Free}(T_\gamma)$ *where* z *is a new variable symbol that occurs in neither sub-formulas.*

The definition of substitution allows deriving several other useful properties and operations. The most straightforward consequence of

this definition is that the meaning of a term does not depend on the naming of the individual parts of the formula but that the computational and grammatical structure defines the meaning in interaction with other terms of the language.

Definition 2.1.6 (α**-conversion**):
Let t *be a* λ*-term of the form* $(\lambda x_\alpha.M_\beta)$ *then*

$$(\lambda x_\alpha.M_\beta) \to_\alpha (\lambda y_\alpha.M_\beta[x_\alpha \leftarrow y_\alpha]) \text{ if } y_\alpha \notin \textbf{Free}(M_\beta)$$

is called α*-conversion.*

With α-conversion it becomes clear that the number of terms with the same grammatical and computational structure is denumerable but infinite in the amount of available names and each term represents a class of similar structure which are equivalent up to renaming of the parts. Another property and a main computational operation is β-conversion which introduces syntactic manipulation commonly used to make the λ-calculus a computational device.

Definition 2.1.7 (β**-conversion**):
The β*-conversion comes in two variations. The first variation reduces the* λ*-binder by one variable and replaces each occurrence of that variable by a term. Formally this is called* β*-reduction and noted as:*

$$(\lambda x_\alpha.S_\beta)T_\alpha \to_\beta S_\beta[x_\alpha \leftarrow T_\alpha]$$

The second variation is called β*-expansion and does the opposite of the reduction: A bound variable is* freed *by adding a parameter to the* λ*-binder replacing the occurrence of the bound term by that parameter and adding the concrete term as the first term in front of the* β*-expanded term such that the application of* β*-reduction will yield the original term.*

The other important operation is the so called η-conversion.

Definition 2.1.8 (η**-conversion**):
Let t *be a term of the form* $(\lambda x_\beta.F_{\beta\to\alpha}x_\beta)$ *where* x_β *is not free in* F *then* F *is the* η*-reduct of* t*. Formally:*

$$(\lambda x_\beta.F_{\beta\to\alpha}x_\beta) \to_\eta F_{\beta\to\alpha} \textit{ if } x_\beta \notin \textbf{Free}(F)$$

The main consequence of these two definitions is a property which is referred to as the Church-Rosser-property [20]. Even though it is only of limited practical importance for this work, another important consequence for automated theorem proving is that **STT** has the strong $\beta\eta$-normalization property.

Definition 2.1.9 ($\beta\eta$**-normalform**):
For every term t *of* ***STT*** *there exists an unique term* t' *such that* $t \to^*_{\beta\eta} t'$ *where* $\to^*_{\beta\eta}$ *denotes the repeated application of* $\to_\beta$ *and* $\to_\eta$ *until it is not further possible.*

With the idea of a normal form and the machinery in place, each derived term can be normalized and many operations need only be defined on normalized terms which helps on the implementation aspect tremendously.

To give meaning to the above defined syntactical structures and operations, an interpretation function into a model has to be defined. The model definition can take quite a few different variations and extensions of **STT** into account depending on which axioms to accept into the systems[2]. This granularity allows studying the needed prerequisites for different problems efficiently at the price of introducing more sophisticated structures like typed applicative structures as an object of study. However, for this thesis only an idea of the underlying formalism and semantics is sufficient since it does not deal with **STT** itself but is more concerned with practical applications which are restricted to a single class of models.

Definition 2.1.10 (**Frame**):
A frame $D = \{D_\alpha\}_{\alpha\in\mathcal{T}}$ *is a collections of sets – sometimes referred to as domains – indexed by types. In correspondence to the base types there is the domain* D_σ *which contains the designated values* $\top$ *and* $\bot$*, a nonempty domain* D_ι*, as well as domains* $D_{\alpha\to\beta}$ *for each freely generated type* $\alpha \to \beta$ *which represent the functions from* $D_\alpha \to D_\beta$.

Definition 2.1.11 (**Interpretation-function**):
An interpretation-function $\mathcal{I}$ *is a function which assigns each constant symbol* c_α *an element in* D_α.

The casually described semantics of the logical connectives of choice can now be given formally:

- The symbols $\top$ and $\bot$ are mapped to their counterparts in D_σ.

$$\mathcal{I}(\top) = \top \quad \text{and} \quad \mathcal{I}(\bot) = \bot$$

- $\mathcal{I}(\neg_{\sigma\to\sigma})$ switches up its input symbol and returns the other from D_σ thus

$$\mathcal{I}(\neg) = x \mapsto \begin{cases} \top \text{ if } x = \bot \\ \bot \text{ otherwise} \end{cases}$$

- $\mathcal{I}(\vee_{\sigma\to\sigma\to\sigma})$ returns true if at least one of the arguments returns true:

$$\mathcal{I}(\vee) = x \ \mapsto y \mapsto \begin{cases} \top \text{ if } x = \top \text{ or } y = \top \\ \bot \text{ otherwise} \end{cases}$$

2 For a deeper dive consider the material by Barendregt [6].

- $\mathcal{I}(\Pi_{(\alpha\to\sigma)\to\sigma})$ returns true if the predicate $p_{\alpha\to\sigma}$ holds for all the inhabitants of the domain D_α.

$$\mathcal{I}(\Pi) = (p_{\alpha\to\sigma}) \mapsto \begin{cases} \top \text{ if } \forall a \in D_\alpha : (pa) = \top \\ \bot \text{ otherwise} \end{cases}$$

The interpretation functions needs to include at least these mappings.

Definition 2.1.12 (**Assignment**):
An assignment @ is a mapping which assigns all occurrences of a variable x_α to an inhabitant of the domain D_α.

Definition 2.1.13 (**Valuation**):
Given a frame D, a signature Σ and an interpretation function $\mathcal{I}$, a valuation V maps a term t_α to elements of the domain D_α such that:

1. $V(x_\alpha, @) = @(x_\alpha)$ *for* $x_\alpha \in \mathcal{V}_\alpha$
2. $V(c_\alpha, @) = \mathcal{I}(c_\alpha)$ *for* $c_\alpha \in \mathcal{C}_\alpha$
3. $V(f_{\alpha\to\beta}\, a_\alpha, @) = (V(f_{\alpha\to\beta}, @)V(a_\alpha, @))$
4. $V((\lambda X_\alpha . s_\beta), @) = f_{\alpha\to\beta} \in D_{\alpha\to\beta}$ *such that each* $v \in D_\alpha$ *is mapped to* $V(s_\beta, @[v \to X_\alpha])$

The pair $(D, \mathcal{I})$ is referred to as an interpretation where $\mathcal{I}$ is required to be a total function.

Based on the notion of a frame, interpretation and assignment it is now possible to define models. An interpretation together with a valuation is a model. Depending on conditions for the frames and the valuation function different semantics can be described. The relevant semantics for this thesis are called Henkin models or general models [38].

Definition 2.1.14 (**Henkin model, General model**):
An interpretation $(D, \mathcal{I})$ is called a Henkin model (General model) if and only if a total valuation V exists which fulfills the properties for a valuation described above. If this is the case, then V is unique.

The main and very important difference to the standard models which historically have been described before – is that the domains in the general model are not required to be full but just need *enough* elements to allow a valuation. Formally: $D_{\alpha\to\beta} \subseteq D_\beta^{D_\alpha}$.

With general models, the level of expressivity of **STT** is the same as first-order logic: It is compact and semi-decidable. Now the syntactical structures can be linked to models. A formula in **STT** is a term t_σ which can have different properties concerning the models which fulfill it.

Definition 2.1.15 (**Satisfaction, Validity**):
Let $\mathcal{M}$ *be a general model and let* @ *be an assignment into* $\mathcal{M}$. $\mathcal{M}$ *satisfies a formula* F *if an assignment* @ *together with a valuation* V *evaluate* F *in* $\mathcal{M}$ *as true. Formally:*

$$\mathcal{M} \vDash_{@} F \textit{ if } V(F, @) = \top$$

F *is valid if for all models* $\mathcal{M}$ *and for all assignments* @*:*

$$\mathcal{M} \vDash_{@} F$$

Usually this is written as: $\vDash F$.

A proof calculus is a collection of formulas and inference rules which can be applied on formulas to make new deductions. Historically the sequent calculus and the calculus of natural deduction from Gentzen [32, 33] and the Hilbert calculus [39] have been important and provide the basis for proof theory[3].

Definition 2.1.16 (**Proof-Calculus**):
Let Γ *be a set of formulas and let* $\mathbf{R}$ *be a set of inference rules then* $p = f_0, f_1, \ldots$ *with is called a proof sequence if for all* f_i *one of the follwing conditions is met:*

- $f_i \in \Gamma$*;*
- f_i *is an axiom;*
- f_i *follows from the application of a rule* $r \in \mathbf{R}$ *to one or more previous formulas* f_j *with* $j < i$.

A proof sequence p *is a proof for a conjecture* C *if* $p_n = C$ *follows with inference rules* $\mathbf{R}$. *This is denoted:* $p_n \vdash_{\mathbf{R}} C$. *The pair* $(\mathbf{R}, \Gamma)$ *is referred to as proof calculus.*

The calculi employed for automated theorem proving are different since the intention is different: the goal is not the formalization of the proof but the efficient mechanization also. Both of the different types of calculi above are obliged to fulfill certain properties in order to guarantee their results and correctness.

Definition 2.1.17 (**Completeness, Soundness**):
Let P *be a proof calculus* $(\mathbf{R}, \Gamma)$. *Then it is called complete if for all* Γ *and* C*:*

$$\vDash C \implies \Gamma \vdash_{\mathbf{R}} C$$

It is called sound if for all Γ *and* C*:*

$$\Gamma \vdash_{\mathbf{R}} C \implies \vDash C$$

3 Here a Hilbert-style calculus is presented because it simpler and sufficient for the purposes of this thesis.

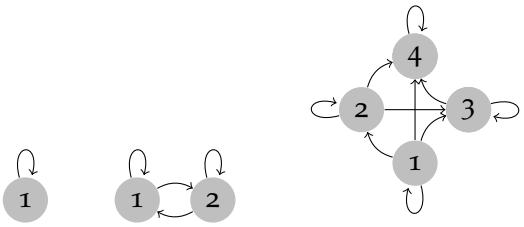

Figure 1: Example of a category with 1 object, 2 objects and a preorder with 4 objects.

The calculus employed in different automated theorem provers differs while maintaining the properties above. As the time has shown – although the calculi are theoretically equivalent – their practical implications lead to different performances depending on the problem domain.

2.2 CATEGORY THEORY PRIMER

Category theory can be regarded as the study of compositions of functions. It provides a very flexible framework under which several fields – starting from numerical fields such as analysis and linear algebra and finishing at logic and type theory – can be abstracted and studied.

Definition 2.2.1 (**Category**):
A category $\mathcal{C}$ *is tuple* $(\mathbf{O}, \mathbf{M})$ *consisting the set of objects* $\mathbf{O}$ *and the set of morphisms*[4] $\mathbf{M}$ *between objects such that the following holds:*

1. $\forall o \in \mathbf{O}\ \exists f \in \mathbf{M} : f(o) = o$
2. *Let* $f \in \mathbf{M}$ *be* $f : a \to b$ *with* $a, b \in \mathbf{O}$ *then domain of* f *– denoted* $dom(f)$ *– is a and the co-domain – denoted* $codom(f)$ *– is b.*
3. $\forall f, g \in \mathbf{O} : dom(g) = codom(f)\ \exists h \in \mathbf{O} : h = g \circ f$. *This composition must satisfy left and right cancellation with the identity e.g:* $id \circ f = f = f \circ id$ *and associativity if all the functions are defined e.g:* $(h \circ g) \circ f = h \circ (g \circ f)$.

The simplest example of a category is the category with only one object 1 and the identity morphism but more complex examples can be easily constructed. For instance, consider the case of a preorder, a set with a reflexive and transitive relation $\leqslant$. By considering the elements of the set $\{1, 2, 3, 4\}$ as the objects of category and defining the existence of an arrow $\to$ between two objects o and o' if and only if $o \leqslant o'$, these two mathematical structures have been identified. Visualizations of these categories can be found in Figure 1.

4 Note that the terms morphism and arrow are used interchangeably.

Often it becomes useful to map a category into another category to either find out that the starting problem exists in different version which may be simpler to solve or to replace concrete values.

Definition 2.2.2 (**Functor**):
Let $\mathcal{C}$ and $\mathcal{D}$ be categories then a functor F between them is a mapping such that

1. *for each object of $\mathcal{C}$ an object in $\mathcal{D}$ is assigned,*
2. *for each arrow in $\mathcal{C}$ an arrow in $\mathcal{D}$ is assigned while preserving identity and composition of arrows:*

$$F(id_{\mathcal{C}}) = id_{F_{\mathcal{C}}} \quad \textit{and} \quad F(g \circ f) = Fg \circ Ff$$

In the tradition of algebra, the same prefixes apply to denote the similar properties. For instance, an endofunctor is an functor $\mathcal{F}$ with $\mathrm{dom}(\mathcal{F}) = \mathrm{codom}(\mathcal{F})$; an autofunctor is an invertible functor and so on.

Categories can not only be mapped to each other, but the functors which exist between the categories can also be mapped. This gives rise to the so-called natural transformations.

Definition 2.2.3 (**Natural transformation**):
Between two functors $F, G : \mathcal{C} \to \mathcal{D}$ a natural transformation μ – or a morphism of functors – is family of morphisms such that

1. *for all $o \in \mathcal{C}$ a mapping $\mu_o : F(o) \to G(o)$ is defined*
2. *for all arrows $a \in \mathcal{C}$ with $a : o \to o'$:*

$$\mu_o \circ F(a) = G(a) \circ \mu_o$$

Definition 2.2.4 (**$\mathcal{F}$-Algebra**):
Given a category $\mathcal{C}$ an $\mathcal{F}$-Algebra is a tuple $\mathcal{A} = (\mathbf{Q}, \alpha)$ where

1. *$\mathcal{F}$ is an endofunctor in $\mathcal{C}$*
2. *$\mathbf{Q}$ is an object in $\mathcal{C}$ often referred to as the carrier,*
3. *$\alpha : \mathcal{F}(\mathbf{Q}) \to \mathbf{Q}$ assigns to each $f \in \mathcal{F}(\mathbf{Q})$ an interpretation $\mathbf{Q}$:*

$$\alpha_f : \mathbf{Q}^{arity(f)} \to \mathbf{Q}$$

.

Definition 2.2.5 (**Initial, Terminal**):
An object $o \in \mathcal{C}$ is called initial if there exists for all objects $o \in \mathcal{C}$ exactly an arrow such that $o \to o'$. An object $o \in \mathcal{C}$ is called terminal if there exists for all objects $o' \in \mathcal{C}$ exactly an arrow such that $o' \to o$.

When considering the nature of functions it is natural to search for special and interesting cases. One of such cases is the case of a fix-point.

Definition 2.2.6 (**Fix-point**):
Let $f : \mathbf{A} \to \mathbf{A}$ *be an endomorphism and let* $p \in A$. *Then* p *is called fix-point of the morphism* f *if and only if*

$$f(p) = p$$

The main application for the fix-point in this work is to consider it for the typing of a recursive tree and to derive the appropriate folding functions based on the initial algebra. As such, the presentation here focuses on the practical rather than the category-theoretic outline. As Meijer, Fokking and Paterson [61] noted recursion operators can be derived from data type definitions reducing the possible recursion schemes significantly and allowing to easily reason about the recursion scheme. In the following the most important schemes for this work will be outlined. A more theoretical founded introduction can be found in the work of Bird and Moore [73].

Definition 2.2.7 (**Catamorphism**):
A catamorphism consumes an inductive data structure – therefore a data-structure defined in terms of an initial algebra – and produces a single output value by repeatedly recombining the individual values generated by a morphism.

As the catamorphism breaks down inductive data structures the anamorphism allows to construct them.

Definition 2.2.8 (**Anamorpism**):
An anamorphism allows to construct a potentially infinite data structure from a single value by applying a funcion to the single value until a given condition for termination evaluates to true.

The classical examples for those particular recursion schemes are the fold and unfold function on lists which can be easily generalized to other initial algebras.

Definition 2.2.9 (**Apomorphism**):
The apomorphism is an extension of the anamorphism which instead of returning a single entry of the data-structure can either return that single entry of the data structure or a more complex inhabitant of the data structure.

2.3 ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Artificial intelligence studies *how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves* [57]. As such, the big goal in the field artificial intelligence is to create a general artificial intelligence that is able to do many tasks at least on human level. Of course, this poses challenges on more than one level since that system cannot resort to do reasoning on only a single strong domain. Systems that restrict themselves to a single domain and achieve strong performances in it, do

not exhibit traits of advanced intelligence and are unable to adapt to other domains, are commonly referred to as weak artificial intelligence[5]. These systems are common today and found for instance chess engines or speech assistant systems like Siri, Alexa or Google Assistant.

In contrast, a strong artificial intelligence is able to exhibit a number of different traits most commonly found by humans. Similar to human beings, the artificial intelligence must be able to represent knowledge in a way so that it can apply it to the given problem. In practice that means to derive judgments from the pool of knowledge, to update this pool of knowledge in such a way that it is consistent when new information from the environment becomes available and to interact with the environment based on the knowledge of the current domain. Furthermore, the ability to communicate with human beings and to plan future actions are considered additional criteria. It is not surprising that once these problems have been formulated a number of research fields have popped up. As examples: Dealing with knowledge – its representation and consistency – is ontology engineering. Dealing with the problem of making judgments with this pool of knowledge is automated theorem proving and for communication and recognition of the current surroundings and objects natural speech recognition, computer vision and natural speech generation are considered sub-fields. Although the results in these fields have been significant, a grand unifying system which comes close to a general artificial intelligence is nowhere to be found so far, since combining these sub fields into a coherent system is a difficult problem. Besides the difficulty of engineering algorithms and approaches, formulating assumptions about how intelligence works and engineering them to technical solutions is an interdisciplinary problem between fields ranging from philosophy to neurobiology and computer science. There are two general ways to tackle the problems in artificial intelligence. Either by using a symbolic approach or a numeric one. The symbolic approach tries to leverage logical methods like described earlier whereas the numeric one tries to solve the by calculation. Recently, the second approach in form of machine learning has shown great results.

In general, machine learning refers to algorithms that learn with experience. As Mitchell [63] noted: *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P improves with experience E*. As such machine learning algorithms are a very interesting field of study because they allow to study the building blocks of intelligence over different expressions and different domains with the effort to formalize them and make them accessible to information processing systems.

5 Of course, these systems are not useless in their respective domains.

The algorithms and methods for artificial intelligence – with the goal of a general artificial intelligence – need to be capable of behavioral self-modification for adaptation to new situations and data – in short a form of introspection with possible adaption to many domains like human beings are capable to. How this introspection will be created is not clear: Some people hope it will magically pop up when enough layers of algorithms and data structures are stacked while others try devise more and more complicated methods by combining various solutions for individual domains into larger and larger systems. However, the seat of consciousness has not been deciphered yet and the machines remain just machines having superhuman performance in certain domains.

The biggest development during the last few years in artificial intelligence has been the development of more efficient training methods of neural networks. Whereas at their inception they have only been theoretical tools due to the inherent inefficiency of training them with the conventional hardware architecture, they have now become the tool of choice over virtually any problem domain in some form or another in artificial intelligence. They allow efficient classification, pattern recognition and embedding which – given the right representation of the problem domain and right input vectors for training – reaches at least human performance on many tasks ranging from drug discovery [56], image recognition [47], and even language translation [86].

In order to construct a general artificial intelligence – therefore a system which is not confined to one special task like a chess engine – quite a few of these tasks have to be solved and what is more important the artificial intelligence has to learn about the world and how to interact with it. The current approach to construct such a system concentrates with great success on applying machine learning and deep learning techniques in every possible domain of artificial intelligence.

There are at least three major and different flavors to machine learning. First, supervised learning, where the algorithm is fed examples of the input and the desired output. The simplest instance of this type of learning problem is the k–nearest-neighbor classification: Given a set $\mathbf{S}$ of labeled objects with features and a function $d : \mathbf{S} \times \mathbf{S} \to \mathbf{R}$ which encodes the distance between two objects of the domain, predict the label of the input object i. A solution is to calculate the distance of each element in $\mathbf{S}$ with i and select the k objects with the least distance to i and let them *vote* with their label. Assign the label to i which got the most votes. In case of a tie, choose randomly. This is what is considered an online algorithm: It can be trained during the execution and the whole system does not need to be turned off. In this instance, the training can just be done by adding the freshly labeled object to $\mathbf{S}$. In contrast, an offline algorithm must cease operations to undergo training.

Second, unsupervised learning where the algorithm is given a way to calculate a global error and it tries to minimize that error by finding patterns in unstructured data. A common instance of this problem is k-means where the algorithm is given a set of unlabeled inputs as well as a distance function – similar to k-nearest-neighbors – and has the task to divide that input into k groups. The common solution is to randomly initialize k objects – which represent the center of the groups – then until the solution stabilizes, assign to each group the object of the input set which has the least distance to that group and calculate afterwards the new center of the group.

A different approach towards the characterization of the learning problem takes the third approach known as reinforcement learning. Whereas the previous approaches assumed some data which encodes information about the domain of their learning task, reinforcement learning characterizes desired and undesired behavior with rewards (punishments) and employs an agent seeking to maximize the rewards to solve the learning problem.

Central for the adaptation of all the approaches is the notion of error or distance: Given the desired outcome, how far is the calculated result of the algorithm away from that solution? The choice of measure of the error depends on the particular domain and on the application of the machine learning algorithm. A common approach is to define the error rate as $\frac{\text{number of wrongly predicted examples}}{\text{total number of examples}}$. However, this is only one measure of error which is of importance. The other one is the generalization error: Since the algorithms are trained on a well-defined training set once they enter the *real world* their performance may suffer because the training set and the learned *rules* do not fit the data good enough and the chosen sample was biased. Choosing an appropriate measure of error is non-trivial since it defines what the system will optimize its behavior against.

Definition 2.3.1 (**Entity, Feature**):
An entity is a collection of quantitatively measured features whereas each feature represents a particular property of the entity which is regarded important for the task at hand.

A large part of efficient machine learning is to recognize the important features of the entities which are present in the input and largely influence the output. However, these features cannot be selected blindly but should be carefully chosen, since each of these features introduces a source of information and leads towards a higher dimensionality of the data.

It is pretty important to notice the distinction between data that is numerical in nature and data that is categorical. The first one is easy to lift into a framework which backed by numerical methods, libraries, and tools. The latter is more difficult to engineer since some sort of numerical abstraction has to be developed first.

Definition 2.3.2 (**One-hot vector encoding**):
Let F *be a set of categorical features with* n *different values and let* $e : F \to \mathbb{N}$ *be an enumeration of those different values. Let* b_n^i *denote a bit vector of length* n *with then* i*-th bit turned on and the rest turned off. A one-hot vector encoding of the categorical feature* $v \in F$ *is given by:*

$$\mathrm{oneHot}_F(v) = b_n^{e(v)}$$

Alternatively sometimes just the enumeration is used which is referred to as just one-hot encoding.

Once the features have been chosen, the data have been acquired and preprocessed, it has be put in the system. A general approach for this is to represent the data as a design matrix. In this representation the data is represented in a matrix $m \in \mathbf{R}^{e \times f}$ where e represents the number of different entities and f represents the different chosen features. This matrix may be extended by another column which contains a label for the entity on that row.

2.4 DEEP LEARNING

Deep learning has dominated machine learning recently and caused a hype cycle in artificial intelligence. Deep Learning or hierarchical learning is the stacking of various neural networks – into hierarchies and combining the individual layers to solve problems. Deep learning builds upon the structures which have been used by neural networks. Those structures in turn imitate the natural biological structures found in the human brain namely synapses, dendrites, axons, and neurons.

In the human brain, the neurons gain their input through their dendrites and if the sum of the incoming signal at the synapse exceeds a threshold it sends out a signal through the axon towards other neurons. From the mathematical perspective, neurons are not freely connected with each other but put into layers. Each of these layers contains a certain number of neurons. Usually, each neuron is connected to all the neurons in the previous layer by directed and weighted links from them to it. These links propagate the activation which is multiplied with the weight of the link.

Definition 2.4.1 (**Neuron**):
Let w^l *denote a weight matrix in the* l*-th layer and let* a^k *denote the vector of activations in layer* k*. Then the activation* a *of a layer* l *can be described as:*

$$z^l = w^l a^{l-1} + b^l$$

$$a^l = g(z^l)$$

where g *is a vectorized activation function,* b^l *is a vector of values referred to as bias and* z^l *is called the weighted input to the activation.* [6].

For a long time the standard function for activation of the neuron has been the sigmoid function.

Definition 2.4.2 (**Sigmoid**):

$$g(z) = \frac{1}{1 + e^{-z}}$$

A classical architecture for neural networks has been the feed-forward network. This architecture aligns multiple layers of neurons: a single input layer followed by an arbitrary amount of so-called hidden layers followed by an output layer. Besides the first layer each neuron in the inner layers is connected to all the neurons in the previous layer. A run of this network consists of setting the input layer to the desired input then the input layer propagates each input through the hidden layers to the output layer by applying the weights, thresholds and activation functions in order to decide whether to trigger or not. Usually each neuron in the input layer represents a feature from the entities under consideration for the algorithm. Meanwhile the values calculated in the output layer are subject to interpretation. From a mathematical perspective, a neural network represents a function which is defined by the sum of its activation functions. With the right neural network configuration and enough training time the function of a neural network can approximate any function.

With the above described approach, it is possible to apply the neural network onto problems. If it is given the same input, the result of the calculation would always return same output since it is not capable of behavioral introspection and self-modification. In order to do so, a training process is used which has the following steps and introduces the notion of back-propagation and error:

1. Initialize the network randomly;
2. Let the neural network make predictions by feed-forward on the data;
3. Calculate the mispredictions the neural network has made;
4. Apply back-propagation to adjust the weights of the network;
5. If the desired level of performance has not been reached go back to 2.

Backpropagation allows adjusting the weights and biases of the layers in the case of an incorrectly predicted output by the network. This requires that the system has a training set with labeled examples

6 The bias is a numerical value to stabilize the triggering process of the neuron and at the same time avoid situations where all the neurons do not trigger.

so that it can determine where it went wrong. For this purpose, an error term or cost function has to be introduced. Depending on the domain and the application many different error-terms can be chosen; for the exemplification of the principles the mean-squared error is sufficient.

Definition 2.4.3 (**Mean-squared error**):
Let i *be the total number of examples in the training set with* t_i *being the label assigned and* y_i *being the value predicted by the neural network. Then the mean squared error of the network can be described as:*

$$\text{MSE} = \frac{1}{2}\sum_{i}(t_i - y_i)^2$$

With the backpropagation algorithm the intention is to minimize the error of the network function by changing the weights of the neurons contributing to the output of the neural network. Therefore, for each layer the contribution towards the error is calculated and then the weights are shifted in the opposite direction. The adjustment of the weights is calculated by taking the derivative, from the output layer and working backwards to the first hidden layer through the set of weights. Let C denote the cost function in the following equations.

Definition 2.4.4 (**Weight-adjustment, Backpropagation**):
Let L *denote the output-layer and* $\odot$ *denote the element-wise multiplication. Then the error in the output layer can be described as:*

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

For any layer $0 < l < L$ *the error is given by:*

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

The derivative for the biases on the same neuron is given by:

$$\frac{\partial C}{\partial b} = \delta$$

The derivative of the cost function C *concerning a weight is then be given by:*

$$\frac{\partial C}{\partial w} = a_{\text{in}}\delta_{\text{out}}$$

With these equations in place the weight-adjustment and bias-adjustment can be given as:

$$w = w - \frac{\partial C}{w} \quad \textit{and} \quad b = b - \frac{\partial C}{\partial b}$$

There are different formulations of back-propagation in a practical setting. The two principal solutions are batch gradient descent

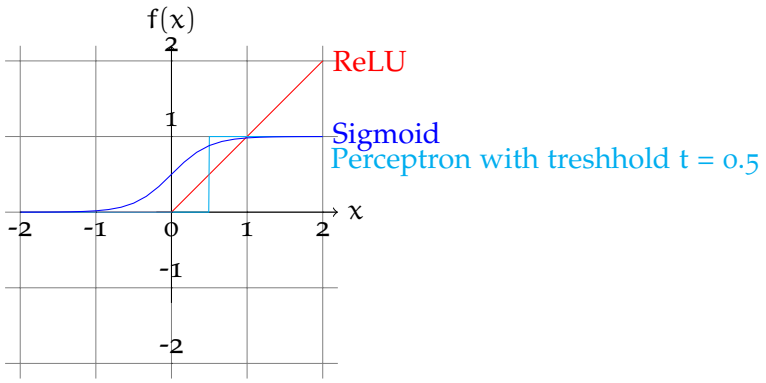

Figure 2: Visualization of different activation functions

and stochastic gradient descent. The difference is when the weight-adjustment is carried out and how the choice of the training instance is done: Batch-gradient-descent adjust the weights based on a whole iteration of the learning algorithm on the data set meanwhile stochastic gradient descent does it after each single prediction on a randomly chosen instance of training data. The compromise is called mini-batch gradient descent which selects fresh subsets of the design matrix and uses them to make a smaller weight-update until the design matrix is exhausted. In practice, this results in compromise between the accuracy of batch gradient descent and the speed of stochastic gradient descend.

Historically, there have been several activation functions used for neural networks. Depending on the data and upon the overall architecture, they must be chosen carefully, since they influence the behavior network immensely by their numerical properties.

The first considered activation function was the perceptron by Pitts and McCulloch [58]; it is rarely used today if at all.

Definition 2.4.5 (**Perceptron**):
Let $t \in \mathbf{R}$ *be a threshold value then the perceptron is given by:*

$$f_t(x) = \begin{cases} \textit{if } x \geqslant t, 1 \\ \textit{else } 0 \end{cases}$$

Besides the sigmoid functions, a very important function for activation is the softmax function. Instead of triggering in a certain way, the function allows to assign an element to a domain from a collection of mutually exclusive domains. Namely, the domain to which this element most likely belongs to.

Definition 2.4.6 (**Softmax**):
Let D *be the number of different domains and* $x \in \mathbf{R}^n$ *then the softmax function is given by:*

$$f(x) = \frac{\exp(x)}{\sum_{j}^{D} \exp(x_j)}$$

Note that $\exp(x)$ *returns the application of the exponential on every element of the vector. Meanwhile* $\exp(x_i)$ *denotes the application of the exponential on the* i*-th element of* x*.*

Another common activation function is the Rectified linear unit (ReLU).

Definition 2.4.7 (**Rectified Linear Unit**):
The rectified linear unit models the triggering of the node if a certain threshold t *has been reached otherwise it just returns the threshold value.*

$$f(x) = \max(t, x)$$

For a visualization of the different activation functions but softmax see Figure 2.

For backpropagation to be applicable, the chosen activation functions in the network have to be differentiable. There are several other architectures for neural networks apart from the feed-forward network described here. They differ in the types of neurons used and allow different kinds of layers and links. However, all of them obey the general principles of operation described here but differ in details concerning activation and training. For an overview consider Liu et al. [53].

Apart from the above examined parameters of the neural network there are so called hyper parameters which allow tuning of the learning process.

LEARNING RATE The learning rate is a factor for the weight-update in backpropagation which determines how strongly the weights are affected by the weight updates. Thus, a learning rate of zero would lead the neural network to ignore the experience it gains meanwhile a learning rate of 1 would lead to jumpy behavior because the most recent experience has the biggest impact on the weights.

REGULARIZATION Regularization controls the size of the weights and thus forces the neural network to make simpler assumptions on the parameters. Practically, this is achieved by adding a term to the error function which relates it to the size of the weights. The two most common regularization methods are l_1 and l_2. l_1 sums the absolute weights and l_2 sums the square of the weights.

MOMENTUM Since the learning algorithm effectively tries to approximate a function, it is sometimes the case that the algorithm gets stuck in a local minimum since the weight updates become very small and the derivations of the functions decrease too. For this reason momentum is used, which carries a small amount from the last weight update over to the current weight update.

Two other common and important issues in machine learning and especially with using neural networks are underfitting and overfitting. Overfitting describes the situation that the algorithm has learned the data so well that it performs with 100% accuracy on the training data but *in the wild* its performance is diminished drastically. Regularization is a way to reduce overfitting. Underfitting describes the other side of the coin: The interpretation of the features is not sufficient for the algorithm to find a correlation which can be exploited for prediction. Machine learning algorithms try to move from a completely underfitted state towards a state which fits the problem well enough in the training, validation set and in real-life applications. Thus, the challenge of is to find the right set of features: Enough to allow generalization but not enough to exactly learn the solutions.

2.5 REINFORCEMENT LEARNING

In 2013 Deepmind [64] has produced an artificial intelligence which has been capable to play old ATARI games with superhuman performance by combining the deep learning architecture with reinforcement learning. Although earlier successes in backgammon [91], chess [45], air hockey [8], financial portfolio management [65] and robotics [90] were already impressive, a new approach which combined deep neural networks with high-dimensional sensory input data appeared, which has been unprecedented.

The general idea in reinforcement learning is to model the program as an agent in a stateful and observable environment which changes with his actions. Then assign desirable actions – actions which lead to an intended result – a reward and punish actions which diverge from the desired result. The fallout is that the agent has to learn a policy for choosing his actions in order to maximize the rewards. The setup is visualized in Figure 3.

Hereby, reward is tightly coupled with the domain the agent operates in. For instance, in games the reward is tied to the result of the game in an obvious way. In portfolio management the reward is tied to the overall value of the portfolio. In robotics the reward is tied to desirable actions of the robot: Moving forward is rewarded and falling is punished. However, there is a difference what the agent is allowed to assume about the world.

For instance in backgammon, the state space is given by the potential states of the board, the distribution of the dices and the potential

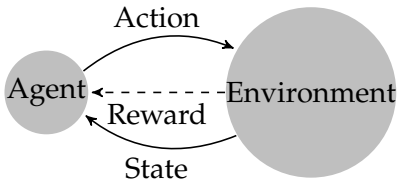

Figure 3: The reinforcement learning environment

moves in each board state. Thus, it can be represented as a very large but finite state space and solved by numerical methods. This referred to as model-based reinforcement learning. The presentation here focuses on model-free learning, which makes no assumptions about the environment the agent operates in and also does not require that the state space is finite. Furthermore, most of the real-world applications fall into this category and this type of learning is also the link where neural networks come to into play. Formally this is described as a Markov Decision Process (MDP).

Definition 2.5.1 (**Markov Decision Process**):
A Markov Decision Process $\mathbf{M}$ *is a tuple* $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \gamma)$ *such that*

$\mathcal{S}$: *is a set of states. A sequence of those states which is indexed by discrete time steps is written as* $(S_t)_{t \in \mathbf{N}}$*;*

$\mathcal{A}$: *is a set of possible actions;*

$\mathcal{R}$: *is an reward function such that* $\mathcal{R} : \mathcal{S} \to \mathcal{A} \to \mathcal{S} \to \mathrm{R}$ *assigns the triple of an action* a *taken in a state* s *and which leads to state* s' *a real value as reward.*

γ: *is a discount factor between* $[0, 1]$ *which encodes the perspective of the agent concerning long-term rewards: A value of* 0 *lets the agent ignore long-term rewards whereas a value of* 1 *let the agent focus solely on the long-term rewards.*

The run or path P of an agent in an MDP can be described as an ordered sequence of tuples $(s_t, a_t, r_t) \in \mathcal{S} \times \mathcal{A} \times \mathcal{R}$ indexed by a discrete time step t.

Definition 2.5.2 (**Cumulative discounted reward**):
Given a run P *of an agent indexed with discrete time steps* t *the cumulative discounted reward from a time step* t *is:*

$$R_t = \sum_{k \geqslant 1}^{P} \gamma^k r_{t+k}$$

The agent is greedy and wants to maximize his rewards and searches for a policy which achieves this goal.

Definition 2.5.3 (**Policy, Optimal Policy**):
The policy $\pi : \mathcal{S} \to \mathcal{A}$ *assigns each state the action to take:*

$$\pi(s) = a$$

Let $Q_\pi(s, a)$ *denote the expected return of taking action* a *in state* s *following policy* π*:*

$$Q_\pi(s, a) = r_t + \gamma \arg\max_a(Q_\pi(s_{t+1}, a))$$

The optimal policy π^* *maximizes the expected cumulative discounted reward from a state* s *by considering the best possible action* a*:*

$$\pi^*(s) = \arg\max_a Q_{\pi^*}(s, a)$$

The Q*-function provides only an estimate of the rewards and has to be updated during training by the rule:*

$$Q_\pi(s_t, a_t) := Q_\pi(s_t, a_t) + (r_t + \gamma \arg\max_a(Q_\pi(s_{t+1}, a) - Q_\pi(s_t, a_t)))$$

When training an agent, there is an area of conflict of interest between the exploitation of the given rules in the current situation and the exploration of different options from the current situation. If the agent just exploits the given policy, then no learning concerning the value of other state-action pairs can take place.

Definition 2.5.4 (ϵ**-greedy policies**):
A ϵ*-greedy policy is a policy* π' *based on a policy* π *such that:*

$$\pi'(s) = \begin{cases} \textit{with probability } \epsilon : \textit{random action} \\ \textit{else } \pi(s) \end{cases}$$

The most common algorithm for *solving* MDPs for the model-free learning problem is the so-called Q-Learning algorithm, which proceeds as follows: First, initialize Q randomly. Then for a defined number of runs called epochs: until a goal state or a fixed depth of evaluation is reached, sample an action a using the ϵ-greedy policy and execute it on the current state s transitioning to the state s' and getting the reward r which is used to update the value of Q for the state action pair as described above.

The link to neural networks is that during infinite state spaces the Q-function cannot be explicitly calculated but only approximated. However, the property of neural networks as universal function approximators comes into play: By defining a numerical representation of the state and the action; and using them as – potentially multi-dimensional – input vectors, they can be fed into the neural network. The architecture of the hidden layers is up to individual design choice but the output layer must consists of a single neuron which activation represents the value the *neural* Q function assigns to $Q(s, a)$.

2.6 THE AUTOMATED THEOREM PROVER LEO-III

Leo-III is the latest incarnation of the Leo-family of automated theorem provers. In the recent CADE ATP SYSTEM Competition (CASC) 2017 of automated theorem provers for higher-order logic, it was the runner-up by solving 382 of 500 problems and only be beaten by the most recent version of Satallax [16]. As nearly all competitive automated theorem provers Leo-III adheres to the TPTP infrastructure and uses the definied THF-format [13] for input as well as the standardized output status ontology SZS [84].

Leo-III uses a cooperative approach [83] by trying to leverage the strength of specialized theorem provers by integrating them apart from having its own higher-order reasoning and proof procedure. This allows Leo-III to be significantly more effective when the problem is inherently in that specialized logic than a pure higher-order system which does not follow this approach.

Furthermore, Leo-III tackles several issues in automated theorem proving which have been neglected. First of all, parallelization of proof procedures has been an ongoing problem in many automated theorem provers. Due to the large theoretical effort in developing an efficient and theoretically complete calculus, the latest trends and progress in programming are not reflected in automated theorem provers. Most of the time, the provers are written in niche languages. In contrast, Leo-III features a modern approach by using the Scala [70] programming language and employing the run-time environment of the Java Virtual Machine (JVM) [52].

Leo-III uses a paramodulation calculus with ordering restraints and a term representation which combines de-Bruijn Indices [24] with a polymorphic representation to express higher-order logic with Henkin-semantics and choice. As such Leo-III tries to deduce the empty clause from the initially given axioms and the negation of the conjecture to prove.

The architecture consists of several modules which together implement the totality of the calculus [14]. An interesting feature to facilitate the construction of proof loops is the existence of a facade[7]. Due to this, it is possible to derive several general proof loops without having rewriting large parts of the system. Therefore, besides the simple sequential proof loop found in most automated theorem provers Leo-III is able to use parallelization as part of the proof process by employing a more complicated state model in form of a blackboard architecture.

For the purpose of this thesis, the sequential proof loop will be used. To structure the search space, it differs between a set of unpro-

7 A facade is a structural pattern in software development which simplifies the access and use of a collection of modules by introducing another module which abstracts out common usage patterns and reduces the amount of operations exposed to the user.

cessed clauses – interchangeably sometimes referred to as working set – and a set of processed clauses.

The *simple* sequential proof procedure follows the following high-level steps:

1. Put all the elements of the input set into the set of unprocessed clauses.
2. Select a clause from the set of unprocessed clause by weighted selection criteria. For this unprocessed clause, apply the inference rules on the clause together with already processed clauses to process it. Each of these operations might result in an inference – a clause or a set of new clauses – which will be simplified and added to the set of unprocessed clauses.
3. Check, if the set of unprocessed clauses is empty or if one of the derived clauses is the empty clause. If so, finish the reasoning process with the appropriate value of the SZS-ontology.
4. Otherwise add the clause to the set of processed clauses and pick another clause to process.

The first formulation of this proof procedure dates back to Wos et al. in 1984 [100] and is quite attractive due to its algorithmic simplicity and effectiveness in practice. At the same time the given formulation makes it difficult to exploit the modern developments of hard- and software like parallelism and concurrent programming. With this procedure in mind, the potential points of research and improvement can be formulated.

Definition 2.6.1 (**Premise Selection**):
Given a set of premises and a conjecture to prove, predict which of the premises will be useful for the proof.

Premise selection or relevance filtering is a well-studied field[8].

It is run at the start of the proof procedure and proposes a binary classification task from a machine learning perspective. Since each included premise from the input file has significant impact on the state space, on the ability of the prover to find the proof goal and sometimes even on the ability to represent the whole input problem efficiently, this is of major concern when dealing with large ontologies of different axioms. When using predefined small problems which only contain the necessary axioms as it is the commonly case in the benchmarks for automated theorem provers, it is of less interest. However, this problem is highly important for interactive theorem provers like Isabelle and Coq, since they reason over large knowledge bases. Premise selection as such can be regarded as a high-level instance of guidance: The prover is not modified but instead a preselection

8 For instance by Meng et al. [62] and Kühlwein [49]

of relevant clauses is done by a preprocessor to help the automated theorem prover achieve the proof goal.

Methods which directly impact the operations of the prover during runtime can be considered low-level instance of guidance: The information for the machine learning algorithm includes the internal state of the prover and the working set:

Definition 2.6.2 (**Proof guidance**):
Proof guidance can be split into two problems:

- *Given the current state of the prover with all the relevant meta information, which clause of the current unprocessed clauses is the best clause to process in order to prove the conjecture? This problem will be named Clause Selection throughout this thesis.*
- *Given the current state of the prover with all the relevant meta information, which available proof operation will be most effective towards reaching the proof goal? This problem will be named Action Selection throughout this thesis.*

However, clause selection is relevant even in the restricted case of small problems since it deals with the traversal of the search space. As such, it must not only binary classify the premises but must also find a ranking in terms of their predicted utility towards the proof goal.

A form of proof-guidance is the so called strategy mechanism which is built into automated theorem provers in various forms. In the simplest implementation, it is a bias on the selection of the next clause by adjusting the weights of the different selection criteria in a favorable fashion for the proof goal. In the more sophisticated variations of the implementation it also includes a bias towards proof operations which are considered unfavorable for the average problems but are particular effective for the requested proof goal. On the scale between high-level guidance and low-level guidance, it is somewhere in the middle.

3 TOWARDS SMARTER AUTOMATED REASONING

3.1 THE STATE OF THE ART

Although the field of automated theorem proving has been around a long time, work on the intersection of numerical and symbolic reasoning has only taken off in the recent years[1] due to the general availability of machine learning libraries and their possible integration into various programming languages as well as the rich ecosystem of possible models for neural networks. Additionally, many issues concerning the interaction and standardization of automated theorem provers were mitigated allowing the integration into new and larger systems. Part of the problem of automated theorem proving is that it is a task which is inherently bound by computing power as well as the memory limitations. Since both of these resources are now abound available, it becomes reasonable to compromise on raw computing power to increase the heuristics used during the proving process.

Historically, Ertel and Suttner [87] explored in 1990 approaches of neural learning to recognize features for clause selection and generate biases towards certain features of the clauses as part of the first-order logic automated theorem prover sequential theorem prover (SETHEO) in order to improve its performance and clause selection. Eventually, the SETHEO-project was discontinued and their generation method was not adapted by other provers.

Presently, most notably for higher-order reasoning has been Machine Learning Strategies (MaLeS) [48] which effectively works as a preprocessor for numerous theorem provers. During a training phase, MaLes approximates effective parameters of the theorem prover for solving the given problems. Depending on the results of the prover, the approximation function for the parameters is adjusted. It has been very effective with the automated theorem prover Satallax where it was capable to leverage the abundance of built-in heuristics effectively into better performance by proposing an optimized schedule of these modes to run. In the end – even though this approach has been effective and is generally reusable to other provers – it has not the marks of a smarter solution towards proving a problem. The author of MaLes indicates, that all of the proofs would have been found anyway if the right mode is provided by the user. Only the knowl-

1 Most notably has been the creation of a dedicated conference which explores applications of machine learning in automated theorem proving: http://aitp-conference.org/

edge of the user involving the choice has been automated and thus the speed of convergence towards a solution has been improved. The proof process was left the same besides being called with the right parametriziation which guided the operations.

Furthermore, there is The Blind Strategymaker (BliStr) by Urban[94]. *BliStr* is a tool which generates strategies for the automated theorem prover E [79] by defining a test-bed of easy problems on which new strategies are generated and testing the fitness of these functions in a pool of slightly more difficult problems. The fittest of these strategies will be chosen to be further improved and put again in the test-bed for repetition. This approach has improved the performance of E in the CASC@Turing competition by 50 Problems in 2012. BliStr has the advantage, that it does not depend on concrete representation of a theory.

As a large company, Google has founded the DeepMath[2] project which is a co-operation between several universities and Google Research to do *experiments towards neural network theorem proving*. As part of that project, Loos et al. [54] have concentrated their efforts on augmenting the first-order prover E with a deep value network for internal proof guidance in the form of clause selection. For this purpose, they have explored different architectures of neural networks which are evaluated with respect to their performance in the Mizar library [93]. All of the three neural networks have improved the performance of the automated theorem prover at the cost of significant overhead due to evaluating the neural network during the proof search for selecting the next clause. They indicate that the evaluation dominates the proof search and that the best results can only be achieved by what they call a hybrid approach: Let the system schedule both – the inbuilt clause selection and the chosen neural network architecture to select the next clause. For running the clause-selection two options are indicated: Either run initially the clause selection of the neural network for half the given timeout and then let the heuristics of E take over and finish the problem or both of the clause selection mechanism in interleaving fashion. The input to the neural networks here has been a string representation of the clause. Interestingly, the character-level encoding of the clause as input into the neural network has been sufficient to determine whether it would be required for the proof goal or not.

Recently Kaliszyk, Chollet and Szegedy [41] published HOLSTEP which is a *machine learning data set for higher-order logic theorem proving* where they formulate applications of machine learning in theorem proving and benchmark potential solutions in interactive theorem provers. Albeit the results are promising and indicate that machine learning techniques have a broad applicability in theorem proving, the focus on interactive theorem provers and their input languages

2 https://github.com/tensorflow/deepmath

makes the transfer to automated theorem provers difficult. Still, the authors describe several different points of application for machine learning algorithms like premise selection and intermediate proof step evaluation and provide deep learning architectures which can be used for this tasks. Again, an interesting result from that paper is that neural networks operating on character-level are good enough to find patterns in premise selection to achieve a significant success rate despite the learning blocks being characters and being only vague faithfully representation.

In the setting of higher-order automated theorem proving, the first significant approach which indicates further refinements of the clause selection has been presented by Färber and Brown[27]. They track the prover state in respect to the declared constants and rank the clauses by using previous runs of Satallax. The authors report a significant increase of proven theorem on the Mizar library of around 26% in comparison towards using the automated theorem prover without this approach. However, the solution is based on hand-engineered features and uses Bayesian reasoning for ranking. For ranking the clauses, their results improved by considering positive and negative examples to gain more information about the search space.

Even more recently Färber, Kaliszyk and Urban [28] presented an approach to bring the fruits of deep learning into the realm of automated theorem proving. By modifying the leancop [71] prover for first-order classical logic and augmenting it with a value network for random search a derivation of leancap was created which is called montecop. The results on their data-sets are promising but are limited by the relative strength of the automated theorem prover on which the system has been developed. Again, the combination of traditional proof-search and deep reinforcement learning performed better than just using one of the two proof-modes.

In 2016, Whalen [98] proposed and implemented a whole deep learning architecture based on Metamath called Holophrasm. The system does not use hand-engineered features for the proof search and features partial proof tree search using tree-based bandit algorithms, deep neural networks for estimating statement provability and sequence-to-sequence models to enumerate its actions. However – although the systems is non-interactive – it is restricted towards the metamath infrastructure and did not achieve state-of-the-art performance on the Mizar benchmarks.

All in all, the number of publications and developments concerning the integration of machine learning, deep learning and reinforcement learning have been steadily increasing. A driving force in that spiking interest is the commercial support by Google.

However, it is yet to be determined in how far the approaches do scale to higher-order logic ATPs and what is exactly best in practice;

especially since the approaches have focused on first-order systems, interactive theorem provers and on premise selection.

Although the neural networks used above in clause selection and premise selection did achieve good results; they rely on a unified or normalized representation of mathematical theories in string representation. The *meaning* of *knowledge* is not captured in a way that is independent of the string representation. To illustrate this problem, consider the case of preorder from the preliminaries (Sect. 2.2). Instead of defining the relation with the symbol $\leqslant$ the same relation could be defined with the symbol κ and the entities renamed to a, b, c and d. A prover with neural network as described in the papers – which was trained on the representation with $\leqslant$ – could hardly generalize its predictions to the new form of expression with κ. These techniques have merit, since they allow tuning of the prover concerning a selected library of mathematical knowledge and gain a better performance from it. Yet, it is not an improvement of the reasoning processes and only highlights what could be done if the internal representation would be better. Constructing an internally better representation which combines numerical as well as symbolic knowledge is difficult and requires combination of several methods currently under research. An outline can be found in Further Work (Sec. 3.7).

Instead of focusing on high-level guidance like premise-selection or strategy selection, this thesis explores how to improve clause selection and action selection for low-level guidance of the prover. As such a concrete optimization towards benchmark is done as little as possible.

3.2 REIFICATION OF THE CONTROL FLOW

In order to make better decisions during the applications of the proof calculus, it is necessary to allow reasoning about the operations and the state of prover. Whereas, the latter is explicitly available, the first is foremost done by the execution environment and implicit. This makes optimizations and reasoning about different options during the proving process impossible. Thus, the operations have to made explicit or reified.

For this purpose, techniques made popular by functional programming will be employed: The so called *free monad* together with the interpreter pattern and generalized algebraic types. The free monad is a technique which has been popularized recently[3] to split between the interpretation and the definition of an abstract syntax tree based on ideas of category theory. It allows to define an object-language

3 To the best of the author's knowledge, the exact point in time when the concept was introduced to programming is unknown. First scientific articles from a mathematical standpoint date back to the 1970's [92, 26]. It is encountered in various blogs from 2010 onward.

and delay the execution until a interpreter is provided. During execution the user may even give different possible interpreters for the object-language such that different results or effects can be generated. The main advantage of this approach is that it allows to capture the language of the problem domain without restricting it to a concrete execution pattern. Therefore, the term of the object language may be used for common techniques of compiler optimization, data flow analysis, conditional rewriting and dependency injection. Furthermore it allows to represent a program as a co-product of its different object-languages and then by giving a interpreter for each term of the object-language it allows to execute the whole program.

In the case of automated theorem proving, the object-language is the proof calculus as it is actually employed within the system. However, there is a choice concerning which operations one would like to encode: The pure rules of the calculus or the methods which are applied by the automated theorem prover. Both of these choices do have advantages and drawbacks.

From a logical perspective, the rules of the calculus are desirable since they are neither concerned with the state of the prover nor its problems. Also the application of the rules is easier to trace and each proof can be directly reconstructed from the proof object. At the same time this advantage is also a disadvantage because the operations of the prover contain much more than just the application of the rules of the pure logical calculus: Mandatory side operations to make the prover effective include indexing, calling external provers, choosing clauses for evaluation as well as strategies for scheduling and ensuring safety properties and invariants to guarantee soundness. All of these operations are not contained in the logical calculus but are found in the methods and functions implemented in the automated theorem prover and largely impact its performance in a practical setting as well as the decision done during the proof process. In the end all methods of the automated theorem prover boil down to operations of the logical calculus plus some extra plumbing to pass the parameters and abstract common sequences of operations of the logical calculus. If the automated theorem prover is regarded as an agent which interacts with its environment and is going to be optimized for proving, every informational tidbit which can be made available, could be important to aid in the proof process. Due to the fact that the second option also simplifies the interpretation of the generated language, it has been chosen.

Since the technical reconstruction of the appropriate data structures contains a lot of boilerplate – which is not suitable for presentation – the presentation here resorts to a high-level approach which omits the concrete technical details but gives an abstract of the construction process.

More formally this equals to the construction of an abstract data type **ControlAlgebra** which encodes the implicit operations of the prover and an injection into this data type during the execution.

Definition 3.2.1 (**ControlAlgebra**):
*Let **Calc** be the set of implicit operations of the prover of the form* $f(x_1, \ldots, x_n)$ *then the reification can be described by the following mapping:*

$$\psi : \textbf{\textit{Calc}} \rightarrow \textbf{\textit{ControlAlgebra}}$$

$$\psi(f(x_1, \ldots, x_n)) = \begin{cases} F(x_1, \ldots, x_n) & \textit{f has result type Unit} \\ F(x_1, \ldots, x_n, f(x_1, \ldots, x_n)) & \textit{otherwise} \end{cases}$$

where F *is a distinguished type of the **ControlAlgebra** with a generated name based on the original calculus operation*[4].

Most notably is the annotation of the newly introduced data type with the result of the function additionally to the parameters.

In order to introduce the side-effect of logging to each operation, a drop-in replacement of the control facade is used. This replacement executes each operation and appends a term of the **ControlAlgebra** to a sequence. At the end of the reasoning process this sequence contains a detailed log of all operations executed.

This allows to employ a similar idea akin to the so called and well-known Curry-Howard-Lambek[5] correspondence and allows to treat the proofs as programs consisting of instructions for proving a theorem out of a set of premises given by the input file. Furthermore, the techniques used in static program analysis can now be employed although some of them are not of interest in the context of automated theorem proving e. g. *Does the program terminate on every input?* or *Are there inputs which lead the program to fail?*. However, other questions have immediate consequences in the resulting proofs e.g *Does the result of the program depend on all the intermediate operations and their results?*. Answering the latter question allows to optimize the steps in the main loop of the automated theorem prover and make them context dependent, strip the resulting proofs of unnecessary steps and even minimize the proofs into just the necessary premises and clauses. Moreover, it allows to identify common patterns over the proofs and generate more efficient proof patterns or *tactics* based on the bare proof calculus.

The next step to achieve the goals above, is to find a mapping m : **ControlAlgebra** $\rightarrow$ **ProofAlgebra**, where the **ProofAlgebra** can be seen as a fix point type of the **ControlAlgebra** with explicit representation of the dependencies of each application of the base calculus rules. However, it will be simpler to present **ProofAlgebra** as

4 Of course this is not the theoretical calculus on which the automated theorem prover is based on but the abstraction used for writing the proving mechanism.

5 For an introduction: [89]

a bottom-up tree automaton[6] since this will define the evaluation semantics *en passant*.

Definition 3.2.2 (**Bottom-up Tree Automata**):
A bottom-up tree automata $\mathcal{A}$ is a tuple $(\mathbf{A}, \mathbf{Q})$ where $\mathbf{A}$ is a $\mathcal{F}$-Algebra together with set of states $\mathbf{Q}$ with designated final states $\mathbf{Q}_f \subseteq \mathbf{Q}$ as well as $|\mathbf{Q}|$ being finite.

For the construction of such an automaton the ground terms need to be defined first. In this case the ground terms are the results of the operations of the calculus which in turn can be used as parameters for the data types which inhabit the **ProofAlgebra**.

Definition 3.2.3 (**GroundTerms**):
*Let **Types** denote the set of types used in **Calc** whether they occur as parameter or result. Then the set of ground terms can be described as*

$$\textbf{\textit{GroundTerms}} := \{\phi(t) \mid t \in \textbf{\textit{Types}}\}$$

where ϕ creates a name distinct from the name of the result type it wraps in order to avoid resolution conflicts. To allow substitution, partial representation as well as evaluation and analysis, the result types are considered optional.

Each of these derived ground terms functions is a wrapper around the result. These generated ground terms are then used as recursion anchors to limit the recursion of the term algebra. In the next step the data types of the **ControlAlgebra** are mapped to new ones which allow a functor definition.

Definition 3.2.4 (**TermAlgebra**):

$$\textbf{\textit{TermAlgebra}} := \{\psi(t(x_1, \ldots, x_n, t(x_1, \ldots, x_n))) \mid t \in \textbf{\textit{ControlAlgebra}}\}$$

where

$$\psi(t(x_1, \ldots, x_n, t(x_1, \ldots, x_n))) = \psi(t(\phi(x_1), \ldots, \phi(x_n))) \text{ and } \phi(t(x_1, \ldots, x_n))$$

The most notable change is subtle: The types of the parameters and result types are changed such that recursion can happen. Now, the implicit computation process for proving theorems has been reverse engineered and thus reified. Finally, it is possible to define the language which allows encoding proofs.

Definition 3.2.5 (**ProofAlgebra**):
*The **ProofAlgebra** is given by the expression:*

$$\textbf{\textit{ProofAlgebra}} := \textbf{\textit{GroundTerms}} \cup \textbf{\textit{TermAlgebra}}$$

*A functor definition for the **ProofAlgebra** is straightforward:*

$$\mathit{map}(t : \textbf{\textit{TermAlgebra}}) = t(\mathit{map}(x_1), \ldots, (x_n)))$$

6 For a complete reference on tree automata, consider Comon [22]; the categorial presentation is taken from http://www.cs.cmu.edu/~neelk/tree-automata.pdf

$$map(g : \textbf{\textit{Ground Algebra}}) = g$$

Now that the representation for proofs is finished a concrete proof sequence must be mapped into the tree representation. Formally, the logged proof operations correspond to a sequence of **ControlAlgebra**-inhabitants and a mapping into the tree automaton must be constructed. The algorithm to do this is based on two main ideas. The first is to apply referential transparency – namely that an expression can be replaced with its value without changing the overall program behavior – in reverse meanwhile the second is to construct the corresponding tree by repeatedly applying substitution. This allows amalgamating matching trees to get a tree matching the symbolic calculation made by the automated theorem prover.

However, the trees are not of uniform shape since they encode the specifics of the proof and vary in depth as well as in the shape of their inner leafs. With the varying shape comes also a varying type for each individual proof such that the creation of an unified execution structure and unified analysis tools are hampered. To remedy this it is necessary to define the trees as a fix-point of the initial algebra generated by the data constructors of the **ProofAlgebra**.

This equals to abstracting out the concrete parameters in the **ProofAlgebra** and replacing them with a type which allows plugging in any other value residing in **ProofAlgebra**. Due to the recursion on the fix-point type it is now possible to regard the **ProofAlgebra** as an abstract syntax tree and formulate all the operations on that tree.

The first step of the algorithm consists in filtering out operations which did not contribute towards the proof. In practice this equals to operations that either did not generate a result or whose generated result was not used for additional operations by any of consecutive steps of the proof.

The next step of the algorithm consists in constructing the elements of the **ProofAlgebra** which directly correspond to operations of the **ControlAlgebra** without added data or control flow dependencies. Thus, it is a simple map over the sequence of **ControlAlgebra** inhabitants with the following function where ϕ denotes mapping into the **TermAlgebra** and ψ denotes the mapping into the **GroundAlgebra**:

$$\begin{aligned} &\text{treeify} : \textbf{ControlAlgebra} \to (\textbf{ProofAlgebra}, \textbf{ProofAlgebra}) \\ &\text{treeify}(x_1, \ldots, x_n, \text{result}) = (\phi(x_1, \ldots, x_n), \psi(\text{result})) \end{aligned}$$

The first element of the tuple represents the operation with the input parameters meanwhile the second element represents the result of the evaluation of the operation. Note that at this step the operation is already a – rather shallow – tree.

Now for the second step, it is important to define an appropriate equality between the result type and the parameters in order to allow the amalgamation to match the reasoning process as closely as

possible. It is straightforward to consider equal instances of the result types to be equal but it does not match the reasoning process entirely. At certain points in the proof process the operations are not on a whole set but on single elements of that set. Therefore, the above definition of equality has to be extended such that a set of clauses and n clause are equal if and only if the clause occurs in that set and is the only clause.

From a principled point of view, this is not optimal since it increases the proofs and loses some faithfulness of the representation for the sake of making the representation easier. But the alternative would be to rewrite the reasoning process in such a way that it is optimal for the mixture of machine learning and symbolic reasoning – a rather complicated task. Especially, given the proving strength of the current version of Leo-III, this is an undertaking which would introduce more errors, problems and weaknesses than things to be gained from doing so.

Furthermore, a practical problem arises during the execution: It is not always clear which of the many potential results is actually important for the computation of the next step. Consider the following scenario: Given to proof rules R_1 and R_2 and a clause d with the relationship $R_1(R_2(d))$ – thus the theorem can be proven by an application of R_1 onto the result of R_2. But the application of R_2 on c does not yield a single clause but a set of clauses $\{a, b, c\}$. In the case of the traditional semantics, this would lead to a choice problem: Which of the clauses to choose to apply R_1 to? There is not enough information to guarantee the choice of the right clause and in the worst case R_2 will be applied to all of them. Furthermore, since the applications of R_2 do not depend on each other, this disadvantage can be easily compensated by concurrently applying R_2 to each of the clauses and then combining the results.

Since the last operations of the reasoning process has triggered prover to find the unit-clause, the construction of the proof tree must happen in reverse by folding the sequence of shallow **ProofAlgebra**-trees from the right. During each step of the fold the accumulator has to be traversed in order find a node in the abstract syntax tree which is equal to the result of the current operation. If this is the case then the node – which has the same result – is replaced with the current operation. Once the algorithm has finished the accumulator contains an tree automaton which directly corresponds to the proof.

From this tree, it is possible to define several useful functions and utilities. The most important one is to generate a signature for the proof such that if the tree automaton is given the input parameters which correspond to the signature is capable of computing the proof.

In order to abstract out the concrete values of the proof, the concrete values of the ground-terms are optional. This allows to replace

Algorithm 1 Constructing the **ProofAlgebra**

```
input := Sequence of ControlAlgebra
for c ∈ reverse(input) do
    if c does not contribute to proof then
        remove c from proof
    end if
end for
proof_object := map(treeify,input)
for p ∈ proof_object with acc as accumulator do
    if acc is empty then
        acc := p
    else
        acc :=merge(acc, p)
    end if
end for
output := acc
```

them by recursive descend with holes which later on can be field by parameters.

As before, this is done by a fold: Recurse into the tree and collect the set of empty leafs. These leafs then can be used as target for a direct replacement with the values needed which is done again by recursive descent into the tree. In consequence the arity of a tree automaton t is a function:

$$\text{arity} : \mathbf{T} \to \mathbf{N}$$

$$\text{arity}(t) = \begin{cases} \text{if } t \in \textbf{TermAlgebra}, t(\text{arity}(x_1), \ldots, \text{arity}(x_n)) \\ \text{if } t \in \textbf{GroundAlgebra} \setminus \mathbf{S}, 1 \\ \text{if } t \in \mathbf{S}, 0 \end{cases}$$

where **S** is a set of special values regarding the arity calculation.

There are two special values and objects for which the substitution and arity must be defined differently: The state object of the prover and the central storage of signatures. Since they encapsulate side-effects of the automated theorem prover – but are part of the evaluation – they must be evaluated without changing the state of the prover or introducing new clauses to the derivation. Therefore, the arity function assigns them the value zero.

Given the arity it is clear that the substitution on trees is a function which takes a list of potential parameters as input and on each encounter of a *hole*, substitutes the matching parameter bar the values from **S**. Let SOME denote the function which – if provided a value of

the **GroundAlgebra** – checks whether the value holds a term for the proof process or not. NotSome is the negation of that function.

$$\text{substitute} : \textbf{ProofAlgebra} \to \textbf{Parameter} \to \textbf{ProofAlgebra}$$

$$\text{substitute}(g)(p_1, \ldots, p_n) = \begin{cases} \text{Some}(g) \to g \\ \text{NotSome}(g) \to p_1 \end{cases}$$

$$\text{substitute}(s(t_1, \ldots, t_n))(p_1 \ldots, p_n) = s(\text{substitute}(t_1), \ldots, \text{substitute}(t_n))$$

It is important to enforce that the substitution operates in a fixed order; thus it traverses the parameter list from left to right. This is important to ensure that the correct parameter value ends up at the correct place.

In order to gain more statistics, each proof is encoded using one-hot encoding: Each operation of the **TermAlgebra** is assigned a natural number i and these numbers are encoded using a bit-set in which the i-th value is one meanwhile the rest is zero. With this method is now possible to define a function which if given a proof tree, is capable of finding out how often each of the rules was used. Note that all the terms of the **GroundAlgebra** are mapped to zero.

$$trace(t(x_1), \ldots, (x_n)) = oneHotVector(t) + trace(x_1) + \cdots + trace(x_n)$$

Furthermore, this idea allows to trace the transitions between the operations efficiently and generate a transition matrix between the different operations by tracking the appropriate values in the recursive structure, transforming them into their numeric equivalents with the one-hot encoding and then incrementing the values in a transition matrix m.

$$\text{transitions} : \textbf{ProofAlgebra} \to \mathbf{R}^{n \times n}$$

$$\text{transitions}(p(x_1, \ldots, x_n)) = \sum_{x=x_1}^{x_n} m(\text{oneHot}(p), \text{oneHot}(x)) + transitions(x)$$

where $m \in R^{n \times n}$ with a 1 at the (n, m) entry.

With the proof as a first class object, it is possible to extract run-time information as well as information concerning the complexity of the proof. Testing the sequential proof loop with logging on various small examples, exposes the computational overhead of the proving sequence: Only about 17% of the generated inferences during the proof loop are useful to determine the end result in the successful case. This poses an additional problem besides the computational overhead and issues with memory consumption: The run-time for several algorithms used in the automated theorem prover deteriorates and leads to time-out – if given – or stalling during proof process.

It is now possible to find common sequences of **ProofAlgebra**-operations. These proof rules will be called compound proof rules and agents interchangeably. This is due to their potential applications: In a sequential proof loop, they could tie several different proof operations together meanwhile if they are formalized in a so called multi-agent system they could be regarded as an agent which becomes active and carries out the operations. The formalization of these compound proof rules in a multi-agent system would also require to introduce additional structures for perceiving information and acting on the state of the prover. Thus the derivation of the compound rules will not take these into consideration but limit itself to the identification of common sequences of proof rules and their extraction.

The extraction itself can use the constructed **ProofAlgebra** and the corresponding tree automata to find out which operations are common. If each found proof is represented as a tree automata then the identification of common proof fragments means to find the common sub-trees over the union of all tree automata.

However, the naive approach is rather difficult and inefficient: For each proof tree consider the set of all sub-trees and find out for each element of these sets how often it occurs. The sub-tree with the most occurrences calculated this way, should be extracted as an individual sequence of operations. Since each single node in a proof tree can be a potential starting point of a sub-tree of that tree and each of those must be compared to other possible sub-trees this quickly becomes infeasible.

The problem of this approach is, that it is – besides being efficient and requiring all the data during the execution – not encoding the length of the **ProofAlgebra** sequence. It may result in just the shallow tree generated in the beginning by the mapping from the **ControlAlgebra** to the **ProofAlgebra** which is undesirable.

Instead this problem can be encoded as a path finding operation using a well-studied approach called the algebraic path problem[7], which studies the way to compute the transitive closure of a matrix over a closed semi-ring.

Definition 3.2.6 (**Semiring/rig**):
A rig[8] *is a structure* $(\mathbf{C}, \oplus, \odot)$ *such that*

1. $\mathbf{C}$ *is a carrier set;*
2. $(\mathbf{C}, \oplus, 0)$ *is an abelian monoid;*
3. $(\mathbf{C}, \odot, 1)$ *is monoid;*

7 This approach has been studied by many computer scientists since the 1950s when it was found out that matrices and graphs correspond. The first entry in a long line of publications about concrete applications was Kleene's algorithm [43] shortly followed by seminal works Floyd [30], Warshall [97], Roy [75]

8 The n is commonly omitted in order to emphasize that it is a ring with negative elements.

4. $\odot$ *distributes over* $\oplus$;
5. $\odot$ *with* 0 *annihilates* $\mathbf{C}$.

By choosing the right semi-ring plenty of problems can be encoded as an algebraic path problem and solved by the same generic algorithms for this purpose. The most relevant for the purposes of this work is the algorithm attributed to Floyd-Warshall which will be written with the notation of the rig in mind:

Algorithm 2 Generalized Floyd-Warshall algorithm

```
input ∈ R^{s×s}
result ∈ R^{s×s}
result := input
for k to s do
    for c to s do
        for r to s do
            result(c, r) := ⊕(result(c, k), result(c, k) ⊙ result(k, r))
        end for
    end for
end for
```

For instance, by taking the rig $(\mathbf{B}, \vee, \wedge)$, whereas $\mathbf{B}$ is the set of Boolean values and considering an adjacency matrix over this rig, the Floyd-Warshall algorithm calculates whether there exists a path between two nodes of the graph. Meanwhile, when choosing the semi-ring $(\mathbf{R} \cup \{\infty\}, \min, +)$ and an appropriate adjacency matrix, the algorithm yields a solution of the classical all-pair shortest path problem. The rig of interest in order to get the transition probabilities is in this case the so called Viterbi-semi-ring $\mathbf{V} = ([0, 1], \max, *)$ which expresses the probability to transition from one operation to the next operation in state-wise fashion and allows to apply maximum-likelyhood reasoning towards the sequence of proof operations.

The tracking of transition produced a matrix $m \in \mathbf{R}^{n \times n}$ for each level of parameters whereas n is the number of operations of the **ProofAlgebra** but in order to get the transition probabilities the columns of the matrix must be normalized.

In order to find common sequences of proof operations over a large number problems, each transition matrix must be made persistent and be stored in a database for further analysis. The data model of each proof consists of a tuple includes of the name of the problem file, a vector which encodes the amount of calls to the different operations of the proof algebra, a matrix which contains the transition probabilities of each operation, as well as the unrefined sequence of control operations, the calculated unified proof as well as an integer to encode the run. In order to produce a new transition matrix, it is sufficient to add all the transition matrices in the database and normalize them. To find a common sequence of steps, it is then sufficient

to use the path reconstruction between all nodes in the graph with a ϵ which represents a threshold value. For this purpose, let w_p be the product of all the probabilities of a path p. At each step of the reconstruction, the weight w of the current path is compared to ϵ. Once it is below that threshold, the path reconstruction for that path is finished.

In order to execute the generated sequences of proof operations an algebra needs to be defined. Similar to the algebras for the transition matrix, this algebra interprets the sequence of proof operations. As earlier indicated, the evaluation proceeds in a fashion similar to tree-automata: It repeatedly reduces the outer-most nodes by interpreting the data in the node in combination with its leafs. However, the algebra interprets the proof operations over the set of the unprocessed clauses and does not differ on their processed state. Furthermore, this execution model never removes clauses from the set of unprocessed clauses and potentially duplicates the clauses since operations are reapplied to clauses multiple times. The effects which involve the update of state and the signature object are regarded as side-effects and not encoded explicitly in the algebra. In the model with the tree automata, the intermediate results of the proof operations are the states **Q** with the set containing the empty clause being the final or accepting state. The defined **TermAlgebra** takes the role of the $\mathcal{F}$-Algebra.

To proof that the representation as well as their interpretation is faithful – in the sense that it will arrive at the same result as the *original* computation – is hard to bring due to the complexities of the programming language, the run-time environment as well as interacting libraries. However, a simple argument for the faithfulness can be made: The construction of the **ProofAlgebra** is based on the definitions and functions present in the original computation and the interpretation acts as a reverse mapping onto the original interpretations of the data structures. In addition to preserving the relevant clauses through the representation – although unnecessary clauses are added – none of the clauses is removed from the set during interpretation even if this means that the working set becomes large. In order to rule out practical problems the **ProofAlgebra** was ran through the test suite of Leo-III. So far no diverging behavior has been found.

With the ability to gain more information about the proof process and generate agent, the proof loop can be analyzed for extension. A simple point of extension is already given with the multi-priority queue which is used to choose the clauses. The multi-queue uses a local approach of ranking which does not use inter-clausal information – thus information which is emergent and cannot be found in a single clause – or a representation of prover state to determine the next clause. The other potential point of extension is the actual choice of a proof operation.

3.3 BREAKING THE TRADITIONAL PROOF LOOP

The standard proof loop from the preliminaries (Sect. 2.6) is simple to implement and effective in application. However, due to the age of the loop it does not accommodate modern developments of hardware and software. For instance, it does not allow to parallelize the processing and generation of clauses or a finer level of granularity when processing clauses. This leads – as already observed – to a large working set where only a small amount of clauses contribute to the overall goal.

Different approaches have been described to address the problem of concurrency and parallelism [15]. A possible solution is what Leo-III's original design would entail: a blackboard architecture where each rule of the proof calculus would be an independent piece of parallel code which could become active on the state of the blackboard then apply its operations and produce a new result which is added to the overall state. However, this would require a fine grained model of parallelism and concurrency concerning the blackboard and the coordination of the agents which at least would require that a local locking mechanism must be in place such that not the whole blackboard is blocked when an agent is active in it. These structures are notoriously hard to get right and hard to get fast and efficient such that they out-perform their sequential counter-parts[9] despite having to do extra state-keeping and operations.

An additional problem of automated theorem proving is that during the proving process it is not clear how to evaluate the state of the prover. In the domain of multiplayer games – where artificial intelligence algorithms are commonly applied – it is often the case that the gain of one player can be equaled to the loss of another player such that the total of all gains and losses equals zero. This in turn allows defining a function for evaluating the current state of game and construct an algorithm or training procedure to create an artificial intelligence which can play the game well. In automated theorem proving a similar evaluation function is hard to define. It is also not clear which exact features are important for the proof process taking place and what an appropriate way of measuring them would be.

What further increases the difficulty of formulating a smarter approach that it is difficult to predict what the consequences of the application of a proof rule will be. In games like chess, go or tic-tac-toe – apart from a clear definition the state of game – the number of actions available in each state over the course of the game form a monotonic decreasing function. Naively, this allows to formulate a search tree for the game by enumerating all the possible actions for each state,

9 Indicators of these problems can be found for instance in the Python programming language [95] which has a global interpreter lock in place and the Ocaml programming language [51] which features a similar mechanism and only recently introduced a stable multi-core branch after years of work [25].

apply them to the state and repeat the procedure for each for states until a result can be derived. The result can take different forms depending on game: In case of tic-tac-toe it is a well-defined end state with either a tie between the players or a win for one of the players. In more complex state spaces which are present in go, chess or automated theorem proving – where the state space is more intricate and has a higher branching factor – such enumeration approaches run into computational boundaries.

Leveraging the already constructed environment for the execution and generation of compound proof rules, it is now possible to put together proving algorithms which do not follow a deterministic structure but choose their actions in a more dynamic fashion. As a consequence of this dynamic environment the state space must reflect the additional complexities which come with a more granular approach to the processing of clauses.

As described in the preliminaries, the proof procedure in Leo-III does track the overall state of clauses and only differs between two states processed and unprocessed and transfers clauses from unprocessed to processed in fixed sequential sequence of operations. This approach is not granular enough to model the different possible states of processing a clause by subset of the proof rules. Naturally, the application of the proof rules concerning a clause forms a lattice for that clause[10].

Definition 3.3.1 (**Lattice**):
A lattice extends a partial ordered set $(S, \leqslant)$ *with to functions* $\oplus$ *and* $\otimes$ *such that*

- *each pair of objects has at most one morphism* $\leqslant$ *which connects them in one direction;*
- *for each pair of objects* (a, b) *exists a smallest possible element* c *such that the join of a and b is bounded from above* $a \oplus b = c$ *with* $\forall ab \exists c : a \leqslant c \wedge b \leqslant c \wedge \forall d : a \leqslant d \wedge b \leqslant d \implies c \leqslant d$
- *for each pair of objects* (a, b) *exists a largest possible object* c *such that the meet of a and b is bounded from below:* $a \otimes b = c$ *with* $\forall ab \exists c : c \leqslant a \wedge c \leqslant b \wedge \forall d : d \leqslant a \wedge d \leqslant b \implies d \leqslant c$
- *there is a terminal object and an inital object concerning* $\leqslant$.

To encode the application of proof rules with each close, once more the one-hot translation of each rule will be used together with an bit-set encoding. The bottom or initial element of the lattice is the bit with all the entries set to zero. The top element which represents the fully processed clause is indicated by a bit-set with a size which

10 Despite being a natural refinement on the processing of clauses and extensive search in the literature, no previous work has been published to the best of the author's knowledge.

equals the number of proof operations. During the execution, each applied operation is translated using the one-hot encoding and the corresponding bit in the bit-set for the clause is switched to 1. In case a proof rule is compound, the reification of the control flow can be used to generate a bit-set which represents the processing state of the clause after application. After application the original bit-set and the bit-set of the agent can be combined with the logical or to represent the new processing state of the clause.

This approach allows to choose and possibly apply only a small number of operations on clause but it comes with the cost of increased and more difficult state keeping. For this purpose an additional annotation for the clauses has to be introduced which tracks the position of the clause in the lattice and thus the state of processing of the clause.

Since equality for the **ProofAlgebra** does not differentiate between a clause and a set containing that clause during execution and does not track the processing state of individual clauses, a mechanism for dealing with the processed clauses has to be introduced.

First, the level of processing has to be detected in the working set of clauses. There are two principal choices:

1. Check after each application of a proof rule synchronously or asynchronously whether the clause is processed now. In the synchronous case this is straightforward to implement but it is not efficient since it includes a complete traversal of the working set after each operation in order to determine which clauses to remove. In the asynchronous case this would lead to a more complex architecture and process model since each clause in the working set must have its own process which fires if the state of the clause reaches the processed state.

2. Delay the checking until an appropriate number of iterations of the proof loop has passed and only then remove all the processed clauses. This approach delays the check and assumes that it will find more than a single processed clause in a single run. However, the disadvantage is that potentially processed clauses are still included in the working set and increase the computational complexity of proof operations without adding new clauses and thus without adding value.

To gain information about the proof process and the operations of the prover reified, a Markov Decision Process can be modeled to apply deep reinforcement learning to action selection without explicit clause selection.

As a state representation the agent uses a combination of statistics defined in the overall state of Leo-III. This includes several numerical statistics already tracked during the proof including the number of processed clauses and unprocessed clauses, the number certain proof

operations have been applied and statistics about the number defined symbols, type constructors and constant symbols. The agent uses a discrete representation of the action state by enumerating all the possible actions using the one-hot encoding for the **ProofAlgebra**. The actions are executed using the earlier described dispatch mechanism for the **ProofAlgebra**.

The rewards and punishments are assigned in the following way:

- The highest reward is given if the agent does prove the theorem. The reward is set to 10000.
- In order to encourage the agent to make progress in the search space, each time it completely processes a clause it gains a reward of 1.
- Every time the agent does nothing[11], it gets punished with a reward of -100.

The neural network to approximate the Q-function has 15 layers consisting of 512 hidden nodes each with the layers regularized. The used regularization is l_2, since it is only applied to a single example. Furthermore the learning constant of the agent is 1, since a found proof should provide a strong reinforcement of the path to it.

Another way to apply reinforcement learning is to learn the priorities of the clause selection mechanism and the subsequent operations of Leo-III.

As described in the preliminaries, Leo-III uses a variation of the given-clause loop in the sequential case. In order to select the next clause to process, Leo-III uses a multi-priority queue with round-robin scheduling. The weight of a priority determines how many clauses are taken with that priority until the next priority is scheduled. This can be optimized by a MDP, too. However, since the weights are the product of seven natural numbers the action space is continious in nature. The state space is a vector $v \in \mathbf{R}^7$. Since the scheduling takes a number of clauses according to those priorities, a discretization to natural numbers reduces the state space significantly. Furthermore, this allows to limit the space of possible actions. Instead of considering gradients and complex reinforcement learning architectures, the actions are just an increment or decrement of each individual parameter. In order to ensure completness, the parameter concerning the oldest clause in working set must remain bigger than one.

The reward structure is as following:

- If the agent finds a proof then it is assigned the reward of 100. Futhermore, a shorter proof indicates better parameters for that problem. For measuering the length of the proof, the number of proof loops needed to arrive at the conclusion is used:

11 In order to keep the one-hot encoding of the **ProofAlgebra** this is necessary.

$\frac{1}{\text{number of proof iterations}}$. This is added as an additional term to the reward.

- If the agent does not find the proof in the alotted time then it is punished with -100;

The setup of the neural network is the same as previosly described, but the discount is set to 0.5.

3.4 EVALUATION

The MDP for action selection described in the previous section has been applied to a test problem of Leo-III which is part of simple tests. As a consequence, this problem is easy to solve. For reference, it is found in Listing 1. The training process was run on a i7-4770K CPU @ 3.50GHz with 16 GB of RAM.

Listing 1: Problem q1.p out of Leo-III's test suite

```
1  fof(a1,axiom, (![X]: p(X))).
   fof(c,conjecture, (?[X]: p(X))).
```

Unfortunately, the MDP is unable to proof the conjecture and does not converge. Although, some learning appears to take place. An indicator of this can be seen in the action choice: The agent starts to avoid doing nothing, since he is punished for it. After tweaking different parameters of the network (increasing the numbers of layers, and the number of neurons in each layer) still no proof was found. After changing the reward structure to a more sophisticated system the behavior improved but still was not able to find a proof. The changed reward structure gives out a small reward (0.5· number of changes caused in the clause lattice by that operation). If the agent completes processing a clause , it gives out a higher reward (10 instead of 1). The agent is punished slightly (-0.1 per clause) for every clause in his working set. These changes to the reward system lead the agent to infer more clauses and guide them better to the processing. However, it was still not able to find a proof. This is probably due to a combination of two different factors:

- The state representation of the prover is not good enough to allow good action selection criteria: It underfits. This can be solved by introducing new and better proxies to represent the overall state. In particular, the suspicion arises the two problems of proof guidance – clause selection and action selection – are two deeply intertwined problems. Solving one of the problems is not possible without addressing the other in a reasonable way.
- The training time was not sufficient. Although a lot of time was allotted for the search, including a more guided approach by

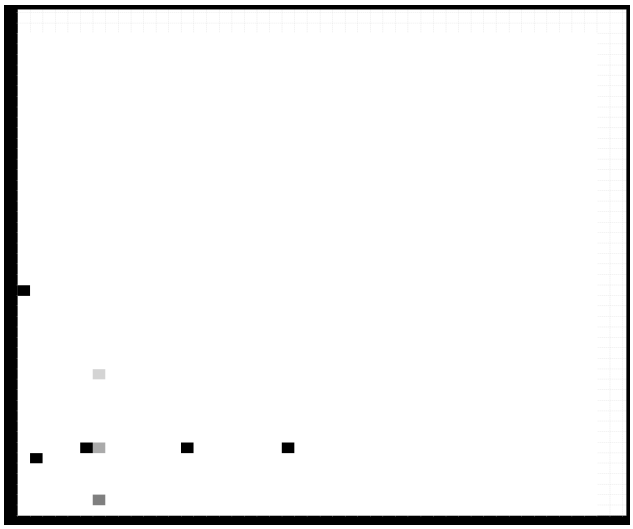

Figure 4: A representation of the transitions for the problem in Listing 1 using a transition matrix interpreted as a heat-map: The darker the values, the more probable a transition the between the states is.

limiting the depth of evaluation, no result was derived. This indicates that in order to foster this approach, a method must be used which employs a semi-supervised technique by replaying successful proof runs within the MDP to reinforce the correct policy. Additionally, a carefully curated set of training problems has to be provided which introduces the common operations of the proof process.

The agent generation could not be evaluated because of problems in the build pipeline and test pipeline. It will be filed subsequently during the oral defense. The general collection of statistics about the proof process and the visualization can be seen in Figure 4 and Figure 5. As is seen in Figure 4 the operations which contribute towards the proof-goal form a sparse transition matrix between the operations. This matrix represents a bias towards certain operations and it should be possible to link the bias towards the domain of the problem and introduce domain-specific inference rules.

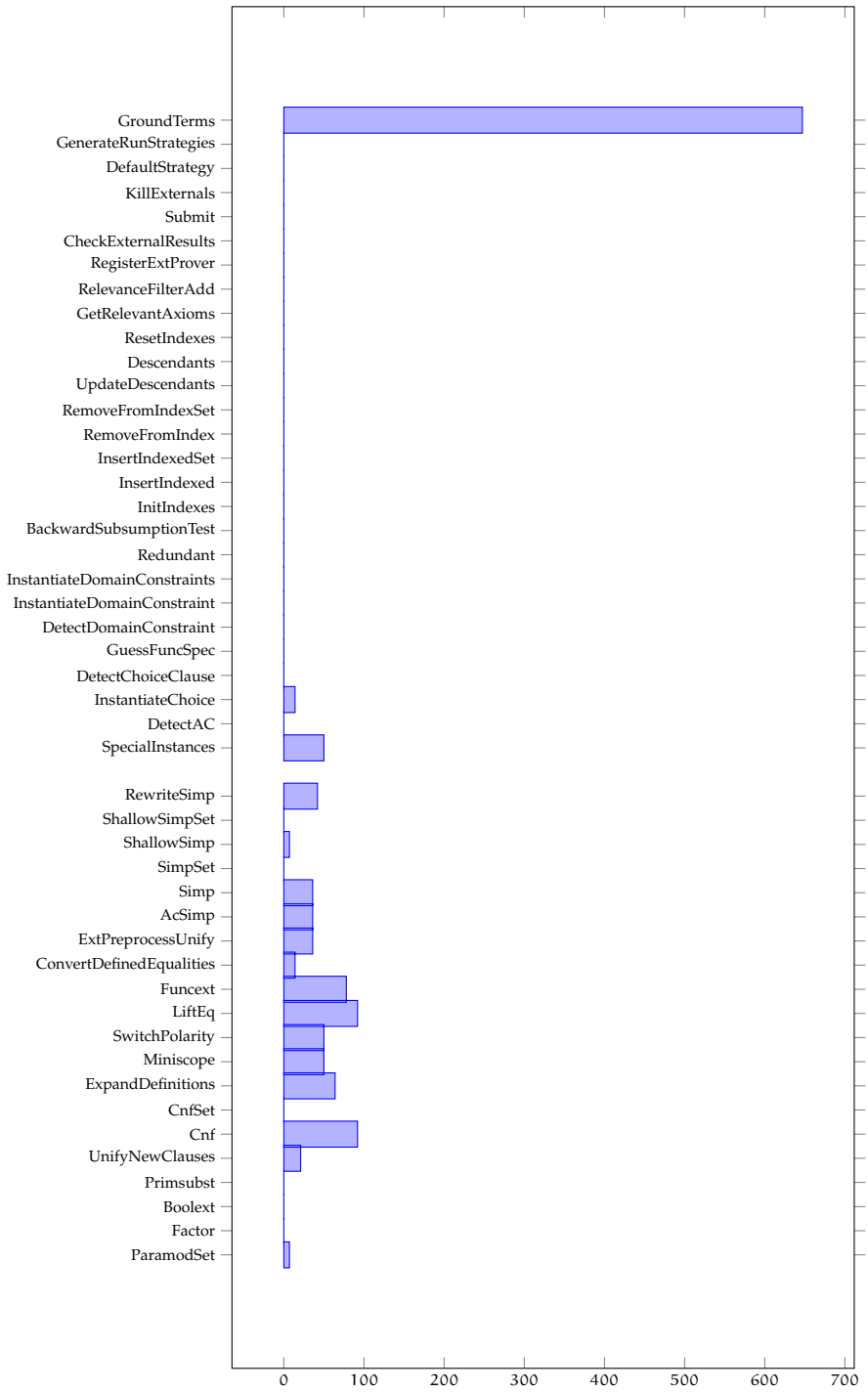

Figure 5: A trace interpreted as histogram without pruned operations for a proof-goal.

3.5 DISCUSSION

Low-level proof-guidance is hard. In contrast to the high-level approaches which have been successfully applied, low-level proof-guidance with deep neural networks requires integration with the whole prover instead of treating the prover as a black box. This includes suitable representation of the set of processed and unprocessed clauses, modifications of the proof-loop and interaction with external libraries.

The last point is especially important, since the internal representation language plays a major factor when it is transformed as an input to the deep learning architecture.

Furthermore, the results achieved in this thesis are limited due to state of the machine learning ecosystem and especially the deep learning ecosystem on the JVM. Whereas other programming languages – especially the lingua franca of machine learning Python – profit from the broad availability of hardware-accelerated deep learning and math libraries; the JVM does not. For one reason or another each of the available frameworks does not fit the bill. Tensorflow[12] uses a static computation graph and applies training and classification during an explicit session which has to be initiated by the user. This makes it cumbersome to integrate and it does not support the neural network architecture described in Further Work (Sect. 3.7). Also, deeplearning4j[13] does not support the needed neural network architectures currently.

Therefore, an effort to integrate the framework *pytorch*[14] into Leo-III has been made during the development of the thesis. It was discontinued since the integration was bad and inefficient. In particular, a problem arises when the state of the prover must be serialized and fed into the machine learning algorithm. Naively, if the machine learning system is not strongly integrated with the automated theorem prover, then the serialization causes the amount of data which represents the state space at least to double. In case of a more sophisticated design – by using a client-server architecture and a custom design protocol which encode the state changes of the prover and sends them over the wire – the state space is still doubled although the exchange of data is reduced[15]. Additionally, non-native integrations for clause selection must replicate the internal structures of Leo-III which is tedious and error-prone from an implementation stand-point.

The integration of *pytorch* to the JVM was done by a tool called *jep*[16]. However, the downside is that the computational model be-

12 https://github.com/eaplatanios/tensorflow_scala
13 https://deeplearning4j.org/
14 http://pytorch.org/
15 This is issue is comparable to the sub-prover integration of Leo-III. Since most the sub-prover do not work with an incremental input stream, the whole state space of Leo-III must be serialized which is computationally expensive.
16 Java Embedded Python

tween Python and the Scala differs in the way things are typed. Meanwhile Scala is strongly typed, Python infamously uses dynamic typing. The data exchange between those systems becomes difficult to get right and efficient, The type casts into the Scala language are problematic. There is no corresponding object for n-dimensional arrays and a parser for the raw string data has to be implemented. In particular, if the machine learning algorithm like MDP uses data structures inside of the main system, the loop of serialization, evaluation, recommendation of an action and feedback becomes awfully slow due to the constant overhead of serialization and reconstruction of the data structures.

The issues concerning the integration of hardware-accelerated neural network appear to settle currently since deeplearning4j is maturing and bindings to Scala of Tensorflow are developed as well as Tensorflow starting to support dynamic computations graphs[17] as well as eager evaluation. In particular, this allows the custom implementation of a reinforcement module for Leo-III powered by one of the above frameworks.

The implementation of the MDP could have been done without implementing the executable proof representation by explicitly one-hot encoding the internal methods of the facade and calling them. But this would limit further experimentation with agents and their optimal application on a proof problem. Also it would prohibit from subsequent experimentation on low-level proof-guidance which needs to include both action selection and clause selection. The proof representation has also the advantage that now the proof loop can be analyzed more thouroghly which gives more insight.

With the theoretical foundations of automated theorem proving worked out good enough to implement systems and deliver results on several domains, the focus has to shift from further development of the theoretical foundations towards the more mundane task of software engineering. Designing the provers not as highly specialized systems but as frameworks with reusable parts to allow recombination with other libraries and integration into larger systems. From all the higher-order automated theorem provers, the approach Leo-III concerning the architecture with the facade, the work on parallelism combined with the a potential integration of machine learning algorithms looks like the most promising.

3.6 CONTRIBUTIONS

The current work has explored the variations and possibilities of low-level proof-guidance by action selection in automated theorem prover for higher-order logic. In the context of the Leo-III project it has contributed an executable proof representation based on category the-

17 https://github.com/tensorflow/fold

ory which can be applied to use techniques for proof compression to generate optimized agents as well as the analysis of proofs by interpreting the proof over a mathematical structure of choice. Given the appropriate but currently missing embedding of formulas and proof rules, this can be used to embed whole proof objects into a vector space. The developed executable representation of proofs is a step into that direction. Furthermore, it allows the identification of unnecessary proof operations, the analysis of the internal data flow of the proof, the creation of proof sequences which exploit these common sequences of operations as well as programmatic manipulation.

The encoding of the proof procedure as a Markov Decision Procedure without effective clause selection has shown – somewhat unsuprisingly – that the internal guidance concerning action selection of an automated theorem prover needs good clause selection. The modeling of the proof process as MDP can serve as a baseline which – once the missing work concerning the clause representation is done – can further be examined and improved. Furthermore, a optimization method for the clause selection process and scheduling based on reinforcement learning has been described.

3.7 FURTHER WORK

With the derived executable proof representation, it is possible to develop Leo-III in multiple directions. A possible extension would be to develop an interactive theorem prover which uses the developed executable representation for tactics. This would also increase the possibilities of training of Leo-III since each decision of the user could be seen as a training situation: The prover predicts the potential choice of the user in the current state.

With the proof representation in place, it is also possible to evaluate a plethora of other machine learning algorithms. For instance, it could be possible to use decision trees by considering different features of the clauses and the current state to determine the effectiveness of an operation and maximize information gain during the proving process. Another option it to apply methods like principal component analysis to the transition graphs. This would yield another perspective on the proof rules and the calculus. Since techniques like principal component analysis allow find the features which carry the most information and rephrase the object under scrutiny using them. Although the interpretation of a numerical value of a proof rule may be difficult[18], atleast it should be possible to determine the *kernel* of a proof – that is the minimal set of rules needed to prove the problem – which could allow a finer grained steps of processing and a potential reorganization of the sequential proof loop.

18 What is 0.5 times the application of a proof rule?

The Bayesian approach that has been applied by Satallax can be expanded. Furthermore, this approach could be combined with the clausal lattice to allow finer grained applications of rules by tracing successful and unsucessful actions during proving besides clause selection.

The MDP for action selection must be improved in order to derive better results. Critical for improvement for this MDP is to implement a solution for representing the set of processed and unprocessed clauses since single action selection – with concurrent application on the working set with eventual removal – is not effective. The working set must be encoded. Therefore, the internal term-representation of Leo-III must be exploited to construct a deep embedding with recursive neural networks[19]. The combination of a tree representation and deep recursive neural networks has been show to be effective in Wang et al. [96]. This would exploit that each term of the λ-calculus has its own *unique* computational structure when represented as an abstract syntax tree. This can be used to construct the embedding into a vector space by recursive descend and recombination of the nodes of the tree representation. Once the recursive neural networks are employed, the proof algebra can be encoded as well.

Another approach could use a hinting technique. This hinting technique should introduce a bias concerning the action selection as well as an bias concerning the clause selection in order to bootstrap the Markov Decision Process faster. The hints should be based on an actual run of the prover on the problem and include positive and negative examples to reinforce the correct behavior.

The MDP for calculating the best weights of the multi-queue priorities can be implemented and tested. If this is successful, it should be expanded to a more complex system which includes all of the internal parameters of Leo-III and optimizes them. Of course, this could introduce further modes for specific types of problems as well, if the reinforcement learning was just applied to a single domain. This would lead to a number of different modes as in other provers.

In October 2017, DeepMind published a massive improvement on their approach to play Go called AlphaGo Zero [81]. Besides reaching the performance of their previous versions in less time, a key feature of the new system is that it did not depend on bootstrapping using human games or small hand-engineered features like the previous version but learned to play unsupervised. Their designed reinforcement algorithm was by a magnitude faster than the previous version of AlphaGo and they claim that it can be generalized towards several domains. Given a clause representation, this appears promising for automated theorem proving. But also just the reinforcement learning pattern could open up further optimizations of the data structures.

19 Other options will run into the representation problem outlined earlier.

The application of non-standard neural network architectures to structure the working set of clauses appears to be promising. In particular, the application of graph neuronal networks as outlined by Kipf et al. [42] since they allow a clause to be regarded as part of an environment. For some problems, the notion of a neighbourhood is relevant since the prover has to guess an appropriate term to solve the problem.

Another application which comes with a better proof representation is to approach the agent generation with genetic programming. To infer useful new agents, the approach by BliStr could be applied in the setting of higher-order logic and on the rule level. The generated compound proof rules can be integrated in the agent structure of Leo-III's blackboard.

3.8 CONCLUSION

Albeit the well-founded theoretical nature of automated theorem provers in general, there are still plenty of new fields to explore. Especially when it comes to integrating machine learning algorithms and approaches into the proving process at a low level. But also the internal representation of proofs as first-class objects inside of the prover opens up new insights into the proof process itself and potential optimization possibilities. In particular, it is possible to judge the used operations and derive new complex operations.

The application of MDP in the presented use-case has failed; but it has been a special use case and could have been expected. Still it provides a baseline to build on and conveyes the idea of an articial intelligence which autonomously tries to prove theorems.

Furthermore, it becomes clear that the given-clause proof-loop employed in several automated theorem provers does not suit the modern computation possibilities since – besides making parallel and concurrent computations difficult – its granularity towards action selection and clause selections leads to the significant increase of the working set. Clause selection together with action selection expands the search space extremely and require more sophisticated methods of structuring the search.

All potential solutions must always be taken with a grain of salt: The problem of providing a proof of a conjecture with a given set of premises is easy if the program is allowed to memorize the solutions and then just return it as answer. Furthermore, if a string representation of the problem is used to the input, the machine learning algorithms do not learn like a human being would. Machine learning solutions contribute towards this memorization in an indirect way. In particular, this makes the results from contests like the CASC harder to interpret and relate to the performance achieved in the real world. Although tuning has been part of the competition, further progress on

the machine learning methods in automated theorem proving could lead to the problem of overfitting described in the preliminaries (Sect. 2.4). Of course even with the generation of strategies, certain properties of the problem domains and problems are already memorized and abstracted out to find convenient preferences concerning the sequence of operations but this effect has not such a strong effect on the competition. Furthermore, due to the state of the provers, these effects are shadowed by general improvements to the prover currently. However, with the ongoing improvement and integration of machine learning techniques into the ATPs, the threat is serious to devalue the competition.

A

APPENDIX

A.1 LIBRARIES

In the course of this worked a lot of code has been written, executed and tested. Meanwhile in the main part I have put focus onto the abstract principles and ideas of the implementation and neglected largely the concrete technical details in favour of streamlining the presentation as well as making the methods clear and applicable to other domains. Furthermore, it would have been impossible to implement all the different libraries in the given time frame myself. Without further ado here is the survery of libraries used.

For the representation and common operations of Matryoshka[1] was used. This library allows to define fixpoints of initial algebras and the correct recursion schemes in a straightforward way.

For visualization and two dimensional computations breeze[2] was used. Breeze offers tools for visualization as well as fast matrix-operations. However, the only type of matrices currently supported is two-dimensional which is limiting.

For deeplearning, deeplearning4j was used. The library is originally Java-library which due to interop can be used by other JVM programming languages. However, the interfaces for other programming languages are to be avoided, since they are badly maintained and barely documented. This applies to the Scala API. Their reinforcement module was used for the MDP-implementation but the experience cannot be recommendated. It is strongly focused on solving the problems on the OpenAI Gym[3] which serves as a comparison and competition environment for reinforcement learning.

1 https://github.com/slamdata/matryoshka
2 https://github.com/scalanlp/breeze
3 https://gym.openai.com/envs/

BIBLIOGRAPHY

[1] Deep RL Bootcamp. https://sites.google.com/view/deep-rl-bootcamp/home. Accessed: 24/10/2017.

[2] UCL Course on RL. http://wwwo.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html. Accessed: 24/10/2017.

[3] Peter Andrews. Church's type theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2014 edition, 2014.

[4] Peter B Andrews. Resolution in type theory. *The Journal of Symbolic Logic*, 36(3):414–432, 1971.

[5] Peter B Andrews. *An introduction to mathematical logic and type theory*, volume 27. Springer Science & Business Media, 2002.

[6] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[7] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

[8] Darrin C Bentivegna and Christopher G Atkeson. Learning from observation using primitives. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 2, pages 1988–1993. IEEE, 2001.

[9] Christoph Benzmüller and Bruno Woltzenlogel Paleo. Automating gödel's ontological proof of god's existence with higher-order automated theorem provers. In *Proceedings of the Twenty-first European Conference on Artificial Intelligence*, pages 93–98. IOS Press, 2014.

[10] Christoph Benzmüller and Bruno Woltzenlogel Paleo. The inconsistency in gödel's ontological argument: a success story for ai in metaphysics. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 936–942. AAAI Press, 2016.

[11] Christoph Benzmüller and Lawrence C Paulson. Multimodal and intuitionistic logics in simple type theory. *Logic Journal of IGPL*, 18(6):881–892, 2010.

[12] Christoph Benzmüller, Chad E Brown, and Michael Kohlhase. Higher-order semantics and extensionality. *The Journal of Symbolic Logic*, 69(4):1027–1088, 2004.

[13] Christoph Benzmuller, Florian Rabe, and Geoff Sutcliffe. Thfo–the core of the tptp language for higher-order logic. *Lecture Notes in Computer Science*, 5195:491–506, 2008.

[14] Christoph Benzmüller, Alexander Steen, and Max Wisniewski. Leo-III version 1.1 (system description). In Thomas Eiter, David Sands, Geoff Sutcliffe, and Andrei Voronkov, editors, *IWIL@LPAR 2017 Workshop and LPAR-21 Short Presentations, Maun, Botswana, May 7-12, 2017*, volume 1 of *Kalpa Publications in Computing*, Maun, Botswana, 2017. EasyChair. URL http://www.easychair.org/publications/paper/342979.

[15] Maria Paola Bonacina. A taxonomy of parallel strategies for deduction. *Annals of Mathematics and Artificial Intelligence*, 29 (1-4):223–257, 2000.

[16] Chad E. Brown. Satallax: An automatic higher-order prover. In *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, pages 111–117, 2012. doi: 10.1007/978-3-642-31365-3_11. URL https://doi.org/10.1007/978-3-642-31365-3_11.

[17] Nikhil Buduma and Nicholas Locascio. *Fundamentals of Deep Learning: Designing Next-generation Machine Intelligence Algorithms*. " O'Reilly Media, Inc.", 2017.

[18] Rudolf Carnap. Abriss der logistik mit besonderer berücksichtigung der relationstheorie und ihrer anwendungen. 1929.

[19] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[20] Alonzo Church and J Barkley Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3): 472–482, 1936.

[21] Leon Chwistek. Über die antinomien der prinzipien der mathematik. *Mathematische Zeitschrift*, 14(1):236–243, 1922.

[22] Hubert Comon. Tree automata techniques and applications. *http://www. grappa. univ-lille3. fr/tata*, 1997.

[23] David Cyrluk, Sreeranga Rajan, Natarajan Shankar, and Mandayam K Srivas. Effective theorem proving for hardware verification. In *Theorem Provers in Circuit Design*, pages 203–222. Springer, 1995.

[24] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[25] Stephen Dolan and KC Sivaramakrishnan. A memory model for multicore ocaml.

[26] Eduardo J Dubuc. Free monoids. *Journal of algebra*, 29(2):208–228, 1974.

[27] Michael Färber and Chad E. Brown. Internal guidance for satallax. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, pages 349–361, 2016. doi: 10.1007/978-3-319-40229-1_24. URL https://doi.org/10.1007/978-3-319-40229-1_24.

[28] Michael Färber, Cezary Kaliszyk, and Josef Urban. Monte carlo connection prover. *arXiv preprint arXiv:1611.05990*, 2016.

[29] William M Farmer. The seven virtues of simple type theory. *Journal of Applied Logic*, 6(3):267–286, 2008.

[30] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[31] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. L. Nebert, 1879.

[32] Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1):176–210, 1935.

[33] Gerhard Gentzen. Untersuchungen über das logische schließen. ii. *Mathematische Zeitschrift*, 39(1):405–431, 1935.

[34] Aurélien Géron. Hands-on machine learning with scikit-learn and tensorflow: concepts, tools, and techniques to build intelligent systems, 2017.

[35] Adam Gibson and Josh Patterson. Deep learning: a practitioner's approach. *To Appear, March*, 2016.

[36] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[37] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi

Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the kepler conjecture. *CoRR*, abs/1501.02155, 2015. URL http://arxiv.org/abs/1501.02155.

[38] Leon Henkin. Completeness in the theory of types. *The Journal of Symbolic Logic*, 15(2):81–91, 1950.

[39] David Hilbert. Die grundlagen der mathematik. In *Die Grundlagen der Mathematik*, pages 1–21. Springer, 1928.

[40] Bart Jacobs. *Categorical logic and type theory*, volume 141. Elsevier, 1999.

[41] Cezary Kaliszyk, François Chollet, and Christian Szegedy. Holstep: A machine learning dataset for higher-order logic theorem proving. *CoRR*, abs/1703.00426, 2017. URL http://arxiv.org/abs/1703.00426.

[42] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[43] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. Technical report, RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.

[44] Donald E. Knuth. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673, December 1974.

[45] Nate Kohl and Peter Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 3, pages 2619–2624. IEEE, 2004.

[46] Boris Konev and Alexei Lisitsa. A SAT attack on the erdos discrepancy conjecture. *CoRR*, abs/1402.2184, 2014. URL http://arxiv.org/abs/1402.2184.

[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[48] Daniel Kühlwein and Josef Urban. Males: A framework for automatic tuning of automated theorem provers. *Journal of Automated Reasoning*, 55(2):91–116, Aug 2015. ISSN 1573-0670. doi: 10.1007/s10817-015-9329-1. URL https://doi.org/10.1007/s10817-015-9329-1.

[49] Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. Mash: machine learning for sledgehammer. In

International Conference on Interactive Theorem Proving, pages 35–50. Springer, 2013.

[50] Joachim Lambek and Philip J Scott. *Introduction to higher-order categorical logic*, volume 7. Cambridge University Press, 1988.

[51] Xavier Leroy. The ocaml programming language. *Online: http://caml. inria. fr/ocaml/index. en. html*, 1998.

[52] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[53] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[54] Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. *CoRR*, abs/1701.06972, 2017. URL http://arxiv.org/abs/1701.06972.

[55] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

[56] David T Manallack and David J Livingstone. Neural networks in drug discovery: have they lived up to their promise? *European Journal of Medicinal Chemistry*, 34(3):195–208, 1999.

[57] John McCarthy, Marvin L Minsky, Nathaniel Rochester, and Claude E Shannon. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. *AI magazine*, 27(4):12, 2006.

[58] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[59] William McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[60] Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232(1-2):91–119, 2000.

[61] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

[62] Jia Meng and Lawrence C Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57, 2009.

[63] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

[64] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[65] John Moody and Matthew Saffell. Learning to trade via direct reinforcement. *IEEE transactions on neural Networks*, 12(4):875–889, 2001.

[66] Reinhard Muskens. Intensional models for the theory of types. *The Journal of Symbolic Logic*, 72(1):98–118, 2007.

[67] Allen Newell and Herbert Simon. The logic theory machine–a complex information processing system. *IRE Transactions on information theory*, 2(3):61–79, 1956.

[68] Michael A Nielsen. Neural networks and deep learning, 2015.

[69] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3-540-43376-7.

[70] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, 2004.

[71] Jens Otten and Wolfgang Bibel. leancop: lean connection-based theorem proving. *Journal of Symbolic Computation*, 36(1):139–161, 2003.

[72] Frank P Ramsey. The foundations of mathematics. *Proceedings of the London Mathematical Society*, 2(1):338–384, 1926.

[73] Oege de Moor Richard Bird. *Algebra of programming*.

[74] Raúl Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[75] Bernard Roy. Transitivité et connexité. *Comptes Rendus Hebdomadaires Des Seances De L Academie Des Sciences*, 249(2):216–218, 1959.

[76] Bertrand Russell. Letter to frege. *From Frege to Gödel*, pages 124–125, 1902.

[77] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27, 1995.

[78] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

[79] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[80] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016. ISSN 0028-0836. doi: 10.1038/nature16961.

[81] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, October 2017. ISSN 00280836. URL http://dx.doi.org/10.1038/nature24270.

[82] David I Spivak. *Category theory for the sciences*. MIT Press, 2014.

[83] Alexander Steen, Max Wisniewski, and Christoph Benzmüller. Agent-based HOL reasoning. In G.-M. Greuel, T. Koch, P. Paule, and A. Sommese, editors, *Mathematical Software – ICMS 2016, 5th International Congress, Proceedings*, volume 9725 of *LNCS*, pages 75–81, Berlin, Germany, 2016. Springer. ISBN 978-3-319-42431-6. doi: 10.1007/978-3-319-42432-3_10. URL http://christoph-benzmueller.de/papers/C56.pdf.

[84] Geoff Sutcliffe. The szs ontologies for automated reasoning software. In *LPAR Workshops*, volume 418, 2008.

[85] Geoff Sutcliffe. The tptp problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337, 2009.

[86] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[87] Christian Suttner and Wolfgang Ertel. Automatic acquisition of search guiding heuristics. In *International Conference on Automated Deduction*, pages 470–484. Springer, 1990.

[88] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[89] Morten Heine B. Sørensen and Pawel Urzyczyn. Lectures on the curry-howard isomorphism, 1998.

[90] Russ Tedrake, Teresa Weirui Zhang, and H Sebastian Seung. Stochastic policy gradient reinforcement learning on a simple 3d biped. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2849–2854. IEEE, 2004.

[91] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[92] Věra Trnková, Jiří Adámek, Václav Koubek, and Jan Reiterman. Free algebras, input processes and free monads. *Commentationes Mathematicae Universitatis Carolinae*, 16(2):339–351, 1975.

[93] Andrzej Trybulec and Howard A Blair. Computer assisted reasoning with mizar. In *IJCAI*, volume 85, pages 26–28, 1985.

[94] Josef Urban. Blistr: The blind strategymaker. *CoRR*, abs/1301.2683, 2013. URL http://arxiv.org/abs/1301.2683.

[95] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[96] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. *CoRR*, abs/1709.09994, 2017. URL http://arxiv.org/abs/1709.09994.

[97] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.

[98] Daniel Whalen. Holophrasm: a neural automated theorem prover for higher-order logic. *CoRR*, abs/1608.02644, 2016. URL http://arxiv.org/abs/1608.02644.

[99] Alfred North Whitehead and Bertrand Russell. *Principia mathematica*, volume 2. University Press, 1912.

[100] Larry Wos, Ross Overbeck, Ewing Lusk, and Jim Boyle. Automated reasoning: introduction and applications. 1984.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of November 19, 2017 (`classicthesis`).