UNIKASSEL
VERSITÄT

Department 16 Elektrotechnik/Informatik
Computer Sciene

Kassel, 26 February 2017

Proof Search in the Sequent Calculus for First-Order Logic with Equality

Master Thesis in Computer Science

Arno Ehle
Mat.-No.: 31216164

Supervisor: Prof. Dr. Martin Lange

Submitted in partial fulfilment of the requirements for the degree Master of Science in Computer Science at the University of Kassel.

Abstract

This thesis introduces an algorithm for automated proof search in the Sequent Calculus for First-Order Logic with Equality (FO[=]). The algorithm is made to be fast on small sequents and to produce small human-readable proofs.
The main idea of the algorithm is to reduce the problem of creating proofs in the Sequent Calculus to multiple validity problems of first-order formulas. These problems can be solved by using an SMT solver for FO[=].
An example implementation is written in Java and has been implemented in the Sequent Calculus Trainer - a desktop application to support students in learning the Sequent Calculus. In the background Microsoft Research's Z3 Prover is used as the SMT solver for FO[=].

Contents

1 Introduction

The Sequent Calculus is a mathematical system to prove that a proposition formulated as a logical formula is always true. From a scientific point of view this is a very important tool when it comes to First-Order Logic. Because of its significance the Sequent Calculus is taught within the lecture *Theoretische Informatik - Logik*[I], which is one of the compulsory lectures for the degree Bachelor of Computer Science at the University of Kassel.

In the past, students had some difficulties in learning the Sequent Calculus [EHL15]. One of the main reasons is a wrong understanding of the syntactic structure of the Sequent Calculus. This problem was targeted and solved with the Sequent Calculus Trainer (version 1.0) – an application for creating proofs in the Sequent Calculus. Figure 1 shows the graphical user interface of the Sequent Calculus Trainer (version 3.0.0).
But students still have trouble in finding correct proofs. You need a deep understanding of the structure and semantic of First-Order Logic[II] to develop ideas on how to prove a sequent[III]. An example for a simple sequent that is difficult to prove is the following one:

$$\forall x.\, P(x) \to P(f(x)) \Longrightarrow \forall x.\, P(x) \to P(f(f(x)))$$

This sequent already requires a lot of rule applications to be proved in the Sequent Calculus and offers a lot of possibilities to apply rules in a wrong way or a wrong order. For beginners it is difficult to master – beside the syntactic structure – the semantics of the Sequent Calculus and thus to find correct proofs. For that reason we decide to made the Sequent Calculus Trainer smart. The aim of this master thesis is to develop an algorithm for automated proof searches in the Sequent Calculus for First-Order Logic with Equality (FO[=]). With such an algorithm the Sequent Calculus Trainer can be improved to help students in finding proofs, to give hints and to create proofs automatically. To achieve this, the algorithm must meet the following requirements:

- The algorithm must solve short examples within seconds. This is necessary for the Sequent Calculus Trainer to give immediate feedback after the user has applied some rules.
- Proofs found by the algorithm must be short and human-readable. It would not be helpful to present a proof containing thousands of steps even for small examples.

In Chapter 2 and Chapter 3 this thesis introduces the concept of First-Order Logic and the Sequent Calculus. The used notation for formulas and sequents refers to the notation used in [LH17], which is also a good starting point to learn the concept of First-Order

[I] Theoretical computer science - logic
[II] See Chapter 2 for an introduction in First-Order Logic
[III] See Chapter 3 to learn more about sequents

Logic and the Sequent Calculus. Chapter 4 is focused on the theoretical groundwork. We will show that the validity problem for propositional sequents with equality is decidable and describe how to obtain proofs for sequents from the decision procedure. For the handling of quantifiers we present an algorithm that uses the reduction of the validity of sequents to the validity of first-order formulas to find applicable instantiations. The reason for using such a reduction is that already a lot of work has been put into various projects focused on the validity problem of first-order formulas. To name just a few projects:

- The *Z3 Prover* by Microsoft Research [DMB08],
- *CVC4* by the Universities of New York and Iowa [BCD+11],
- *E Theorem Prover* by Stephan Schulz (TU München) [Sch13],
- *VAMPIRE* by the University of Manchester [RV02] and
- *SPASS* by the Max Planck Institute for Computer Science [WDF+09].

The Z3 Prover, for instance, is a project that covers 561598 lines of code[I], where the main effort is spent to solve quantified formulas. This is why it would be a good idea to use an existing SMT solver to handle quantified formulas in the Sequent Calculus rather than implement some new sort of quantifier handling from scratch.
The last section of Chapter 4 combines the handling for propositional sequents with equalities and the handling for quantifiers to build a sound algorithm for first-order sequents (the Term Guessing Algorithm). Finally, we will take a look at the Sequent Calculus Trainer (Chapter 6), its implementation of the Term Guessing Algorithm (Chapter 5) and discuss some improvements to further reduce the computation time and the size of generated proofs (Section 5.1, Section 5.2).

[I] Lines of code statistics for the Z3 Prover project on GitHub as of 04-01-2017. Files corresponding to the following set of programming language are considered for the calculation: C++, Python, C#, Java, C, OCaml, CMake, MSBuild scripts, ASP.Net and Bourne Shell.

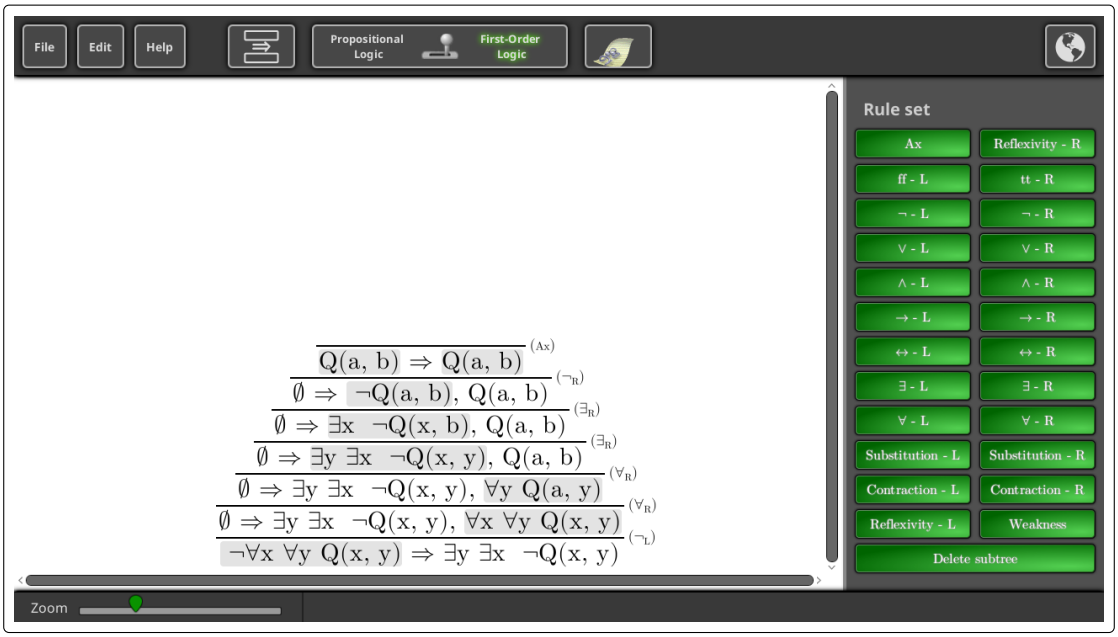

Figure 1: Screenshot of the Sequent Calculus Trainer (SCT) version 3.0.0: showing the SCT with a finished proof.

2 First-Order Logic

First-Order Logic is a well known and common tool in mathematics and computer science. It is used in various parts of these scientific fields and for this reason already part of many textbooks [EFT07] [Fit90]. This thesis is about First-Order Logic over uninterpreted function symbols with Equality. But first of all one needs a notion about what First-Order Logic actually is.

With First-Order Logic it is possible to formulate statements in a mathematically precise way. A statement is called a formula and can be true or false. This depends on the formula and on our interpretation of certain parts of the formula. The following lines are examples for first-order formulas. The idea of these formulas is to connect smaller statements to larger ones using logical connectives like *'and'* ($\wedge$), *'or'* ($\vee$) and *'not'* ($\neg$).

$$A \wedge B \tag{1}$$
$$A \rightarrow \neg B \tag{2}$$
$$\forall x.\, P(x) \vee P(f(x)) \tag{3}$$
$$\exists x\ \exists y.\, \neg x \doteq y \tag{4}$$

2.1 Syntax and Semantic

The whole specification of First-Order Logic covers some more symbols.

- Free and bound variables (e.g. $x, y, z, \dots$)
- Functions (e.g. $f, g, h, \dots$).
- Relations (e.g. $P, Q, \mathit{IsStudent}, \dots$)
- A predefined relation $\doteq$ that is used to make statements about equal terms.
- Logical connectives: the logical not ($\neg$), the logical and ($\wedge$), the logical or ($\vee$), the implication ($\rightarrow$) and the equivalence or bi-implication ($\leftrightarrow$).
- Quantifiers: the existential quantifier ($\exists$) and the universal quantifier ($\forall$)

Definition 1. *Variables and constants are terms. If $t_1 \dots t_n$ are terms and f is an n-ary function symbol, $f(t_1, \dots, t_n)$ is also a term.*

Infinite function applications are not allowed. E.g. $t_\infty \coloneqq f(t_\infty)$ is not a term. Some examples for valid terms over $\{c^{(0)}, d^{(0)}, f^{(1)}, g^{(3)}\}$ as the set of available function symbols are: c, $f(c)$, $g(c, c, c)$, $g(f(c), d, f(f(c)))$.

Based on Definition 1 a term always represents a tree of function symbols, where nodes are made out of the used function symbols and edges are used to indicate the connection between a function and its arguments. Figure 2 illustrates the symbol tree for the example term $f(g(c, d, f(c)))$. The arguments of a function application are an ordered sequence, so the tree representation for a given term t is clearly defined.

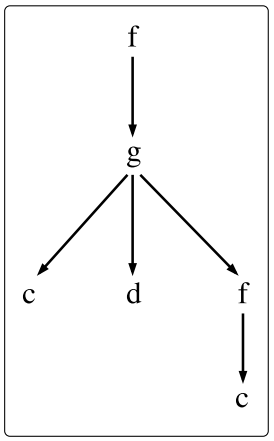

Figure 2: The tree representation of the term $f(g(c, d, f(c)))$

Each node could be identified using a special naming convention based on the position in the tree. The root node has the position ε and if some node n_C is the i'th child of the parent n_P with position p, then n_C has the position $p + ':' + i$.

The term shown in Figure 2 has the subterm $f(g(c, d, f(c)))$ at position ε, $g(c, d, f(c))$ at position $\varepsilon{:}1$, c at position $\varepsilon{:}1{:}1$, another c at position $\varepsilon{:}1{:}3{:}1$, d at position $\varepsilon{:}1{:}2$ and finally $f(c)$ at position $\varepsilon{:}1{:}3$.

A subterm is a term that is part of (is contained within) another term. If a term t_s is a subterm of t at the position p, we write $t_s \Subset_p t$ or $t \ni_p t_s$.

A term substitution is a replacement of one subterm by another or an insertion of a term within a context. The notation for a substitution step is $t[t_{sn}/t_{so}]_p$ or $t[t_{sn}]_p$, where t_{sn} is the new subterm at position p in the term (or context) t and t_{so} is the previous subterm at position p. A term with a missing subterm is called a context – a term with a hole. The underscore sign is used to denote places with a missing subterm. E.g. $f(g(c, \_, f(c)))$ is the context for the subterm d in the term $f(g(c, d, f(c)))$ and $f(\_)$ would be the context of $g(c, d, f(c))$ in this example.

Sometimes it is necessary to replace all occurrences of the same subterm by another one. This is especially the case when we want to instantiate bound variables. We write $\varphi[t/x]$ – without a position – to express that all occurrences of the term x in φ are replaced by the term t.

Definition 2. *The nesting depth $\mathcal{N}(t)$ (or height of a term t) is a number that characterizes the longest path in the tree representation of the term t and is defined as:*

$$\mathcal{N}(t) \coloneqq \begin{cases} t \text{ is a constant}: & 0 \\ t = f(t_1, \ldots, t_n): & 1 + \max\{\mathcal{N}(t_1), \ldots, \mathcal{N}(t_n)\} \end{cases}$$

The nesting depth is an important property of a term because of the inductive definition of terms and will be used in this thesis for some proofs and algorithms. Due to the fact that infinite function applications are not allowed the nesting depth $\mathcal{N}(t)$ is always well defined by a natural number.

Definition 3. *If $R^{(n)}$ is an n-ary relation symbol and $t_1, \ldots, t_n$ are terms, then $R(t_1, \ldots, t_n)$ is a formula. If t_1 and t_2 are again terms, then $t_1 \doteq t_2$ is a formula and if φ and ψ are formulas, then*

- $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \rightarrow \psi$ *and* $\varphi \leftrightarrow \psi$ *are also formulas,*
- $\exists x\ \varphi$ *and* $\forall x\ \varphi$ *are also formulas, where* x *becomes a bound variable.*

What is missing to make the definition of First-Order Logic complete is its semantics. In general, first-order formulas are evaluated to a Boolean[I] value (*true* or *false*). But to do so it is necessary to offer a definition for all symbols used within the formula first.

Definition 4. *A signature* τ *is a list of all available relation* $(R_1, \ldots, R_n)$ *and function symbols* $(f_1, \ldots, f_m)$*. Furthermore, the signature defines the arity* $a_1, \ldots, a_n$ *of every relation and function* $a'_1, \ldots, a'_m$*.*

$$\tau \coloneqq \langle R_1^{(a_1)}, \ldots, R_n^{(a_n)}, f_1^{(a'_1)}, \ldots, f_m^{(a'_m)} \rangle$$

A signature for a specific formula φ must contain at least all used relation and function symbols in φ but it may define additional unused symbols.

Definition 5. *A structure* $\mathcal{S} \coloneqq (D, R_1^S, \ldots, R_n^S, f_1^S, \ldots, f_m^S)$ *is a tuple containing a finite or countably infinite set of elements* D *(the domain) and a definition for all relation and function symbols given by a signature* τ*. If* $\mathcal{S}$ *is a structure over the symbols given by the signature* τ*, we say* S *is a* τ*-structure.*

Please note that R_i^S is the set of mappings that defines the semantic for the symbol R_i and R_i is just the symbol that may be used within the formula. Multiple structures for the same signature may offer different definitions for R_i. The same holds for function symbols.

Definition 6. *An interpretation* $I_\varphi \coloneqq (\mathcal{S}_\tau, \mathcal{V})$ *for a formula* φ *is a pair of a structure* $\mathcal{S}_\tau$ *and a mapping function* $\mathcal{V}$*.* $\mathcal{V}$ *associates to every free variable an element from the domain* D *in* τ*. In other words,* $\mathcal{V}$ *defines the variable assignment used by the interpretation* I*.*

Now, with an interpretation I it is possible to define what it means to evaluate terms and formulas.

Definition 7. *Let* $I \coloneqq (\mathcal{S}, \mathcal{V})$ *be an interpretation and* $\mathcal{S} \coloneqq (D, R_1^S, \ldots, R_n^S, f_1^S, \ldots, f_m^S)$ *be the corresponding structure. A term* t *is then evaluated to an element* $e \in D$ *by using the definition for variables and functions.*

$$[\![t]\!]_{\mathcal{V}}^{\mathcal{S}} \coloneqq \begin{cases} t \text{ is a variable} & \mathcal{V}(t) \\ t = f_i(t_1, \ldots, t_o), 1 \leq i \leq m & f_i^S([\![t_1]\!]_{\mathcal{V}}^{\mathcal{S}}, \ldots, [\![t_o]\!]_{\mathcal{V}}^{\mathcal{S}}) \end{cases}$$

[I] The truth values true and false are called Boolean values. They are named after the mathematician George Boole.

Definition 8. *An interpretation $I \coloneqq (\mathcal{S}, \mathcal{V})$ satisfies a formula $\hat{\varphi}$ ($I \models \hat{\varphi}$) if $\hat{\varphi}$ is evaluated to true using the interpretation I. Let D be the domain of $\mathcal{S}$ and φ, ψ be formulas.*

$$I \models \hat{\varphi} \quad \textit{iff} \quad \begin{cases} \hat{\varphi} = R(t_1, \ldots, t_n) & (\llbracket t_1 \rrbracket^{\mathcal{S}}_{\mathcal{V}}, \ldots, \llbracket t_n \rrbracket^{\mathcal{S}}_{\mathcal{V}}) \in R^{\mathcal{S}} \\ \hat{\varphi} = t_1 \doteq t_2 & \llbracket t_1 \rrbracket^{\mathcal{S}}_{\mathcal{V}} = \llbracket t_2 \rrbracket^{\mathcal{S}}_{\mathcal{V}} \\ \hat{\varphi} = \neg\varphi & I \not\models \varphi \\ \hat{\varphi} = \varphi \wedge \psi & I \models \varphi \textit{ and } I \models \psi \\ \hat{\varphi} = \varphi \vee \psi & I \models \varphi \textit{ or } I \models \psi \\ \hat{\varphi} = \varphi \rightarrow \psi & \textit{If } I \models \varphi \textit{ then } I \models \psi \\ \hat{\varphi} = \varphi \leftrightarrow \psi & I \models \varphi \;=\; I \models \psi \\ \hat{\varphi} = \exists x\, \varphi & \textit{If there is a } d \in D \textit{ so that } (\mathcal{S}, \mathcal{V}[x \mapsto d]) \models \varphi \\ \hat{\varphi} = \forall x\, \varphi & \textit{If for all } d \in D\ (\mathcal{S}, \mathcal{V}[x \mapsto d]) \models \varphi \end{cases}$$

The evaluation order of formulas and terms must be specified with brackets – for example $(A \vee B) \wedge C$. But for this and other reasons formulas may become hard to read. To improve the readability of formulas we make the following definitions:

1. To reduce the number of brackets, this thesis uses $\neg, \exists, \forall, \wedge, \vee, \rightarrow, \leftrightarrow$ as the order of precedence, where $\neg$ is evaluated first and $\leftrightarrow$ is evaluated last. For example, $\neg A \wedge \exists x\ \forall y\ R(x, y) \leftrightarrow C$ is equivalent to $((\neg A) \wedge (\exists x\ (\forall y\ R(x, y)))) \leftrightarrow C$. Multiple operators of the same kind are evaluated from left to right. $A \wedge B \wedge C \wedge D$ is equivalent to $((A \wedge B) \wedge C) \wedge D$.

2. A dot can be used instead of brackets. The dot covers all formulas starting from the position of the dot sign up to the closing bracket of the nearest surrounding pair of brackets. If no surrounding pair of brackets exists, it covers the whole formula starting from the position of the dot. For example $A \;\wedge\; \exists x.\ P(x) \vee Q(x) \vee B$ is equivalent to $A \;\wedge\; \exists x \;\; (\; P(x) \vee Q(x) \vee B \;)$ and $(A \wedge \exists x.\ P(x) \vee Q(x)) \vee B$ is equivalent to $(A \wedge \exists x \;(\; P(x) \vee Q(x)) \;) \vee B$.

3. This paper does not use free variables. New constants will be used instead, which serves the same purpose. A term that has no free variables is called a *ground term*. A formula that only contains bound variables is called a *sentence*. Consequently an interpretation for a sentence has no variable mappings: $I = (\mathcal{S},\ \emptyset)$.

4. A symbol must not be used for multiple roles. E.g. it is not permitted to use a symbol x as a bound variable and as a constant in the same formula. The formula $\forall x\ P(x) \leftrightarrow Q(x)$ is, for example, not allowed because the symbol x is used as constant in $Q(x)$ and as bound variable in $P(x)$ (see item 1 for implicit brackets). Such formulas are confusing to the reader and this is why we avoid double usages.

5. Relation symbols must start with an uppercase letter, whereas function and vari-

able symbols must start with a lowercase letter. In addition, the symbols a, b, c, d and e are always used as constant function symbols, g and h are always function symbols with an arity of one or greater and x, y, z are always used as bound variable symbols.

6. Brackets for parameter lists of 0-ary relations such as $A()$, $B()$ or $C()$ and 0-ary functions (constants) such as $a()$, $b()$ or $c()$ will be omitted: A, B, C, a, b, c.

7. The symbol tt is used as the Boolean constant for true. It is defined as $A \vee \neg A$. The symbol ff is used as the Boolean constant for false. It is defined as $A \wedge \neg A$.

2.2 Satisfiability and Validity

If a formula φ is evaluated to be true using the interpretation I, we say I is a model of φ or I satisfies φ or just $I \models \varphi$. If such an interpretation exists – no matter which domain is used and how symbols are defined – for a formula φ, then φ is called satisfiable.

Definition 9. *Let φ be a formula.*

- *φ is satisfiable iff an interpretation I exists so that $I \models \varphi$*
- *φ is unsatisfiable iff $I \not\models \varphi$ for all interpretations I*
- *φ is valid iff $I \models \varphi$ for all interpretations I*
- *φ is invalid iff at least one interpretation I exists so that $I \not\models \varphi$*

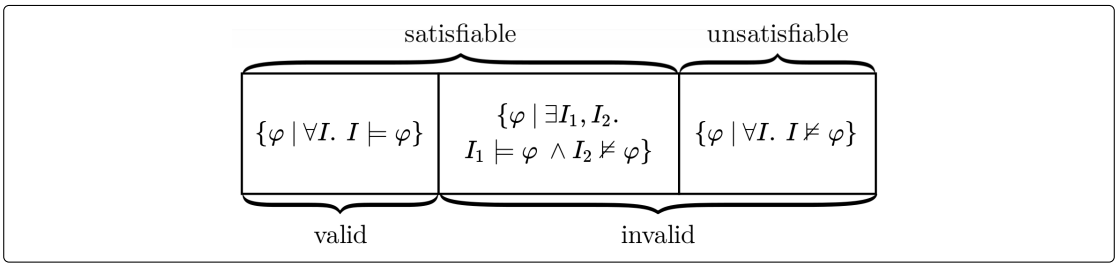

Figure 3: Overview diagram showing the different meanings of satisfiable, unsatisfiable, valid and invalid.

To clarify: the opposite of valid is invalid and the opposite of satisfiable is unsatisfiable. But the negation of a valid formula is unsatisfiable and vice versa, while the negation of a satisfiable formula is invalid and may still be satisfiable as well.

It is not decidable if a given first-order formula is unsatisfiable [EFT07, Section 10.4]. This is proved by the reduction of the *halting problem* to First-Order Logic. However, it is semi-decidable which is a conclusion of Herbrand's theorem [EFT07, Section 11.1]. A

well known semi-decision algorithm based on Herbrand's theorem is Gilmore's Algorithm [Gil60], which terminates for a given formula if it is unsatisfiable.

3 The Sequent Calculus

The previous section introduced the notion of satisfiability and validity for formulas, but the remaining question is: how to prove if a formula is satisfiable or valid? Proving the satisfiability of a formula φ is easy in the sense that it is sufficient to create one interpretation I that is a model of φ. To prove the validity of a formula one needs to show $I \models \varphi$ for *all* imaginable interpretations. This is why we need a sound and complete technique to create proofs. Another important question is: does one formula result from another? Let φ and ψ be formulas: is every interpretation I that is a model of φ also a model of ψ? Both problems can be reduced into each other and thus are equivalent. If φ follows from the formula tt, it is valid. If the formula $\varphi \rightarrow \psi$ is valid, then ψ follows from φ.
The Sequent Calculus targets this type of questions using sequents over uninterpreted functions. A sequent $S \coloneqq (\Gamma, \Delta)$ is a pair of formula sets written as $\Gamma \implies \Delta$. The first set is called the *antecedent* and the second one is called *succedent*. Due to the syntactical structure of a sequent, the antecedent is also called *the left side* and the succedent is called *the right side* of a sequent.

Definition 10. *If an interpretation I is a model for all formulas in a formula set Φ, we write $I \models \Phi$.*

Definition 11. *A sequent $\Gamma \implies \Delta$ is valid, if for all interpretations I the following applies: if I satisfies all formulas in Γ, I satisfies at least one formula in Δ.*

$$\Gamma \implies \Delta \quad \textit{is valid, iff} \quad \forall I.\, I \models \Gamma \rightarrow \exists \varphi \in \Delta\; I \models \varphi$$

With this definition the antecedent is interpreted as a conjunction. "*If I satisfies all formulas in Γ*" can be written as "*If I satisfies $\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n$ with $\varphi_1, \ldots, \varphi_n \in \Gamma$*". In contrast, the succedent is interpreted as a disjunction. "*I satisfies at least one formula in Δ*" can be written as "*I satisfies $\varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_m$ with $\varphi_1, \ldots, \varphi_m \in \Delta$*". For this reason, the following two sequents are equivalent:

$$A \wedge B \implies C \vee D$$
$$A, B \implies C, D$$

Some shorthand notations used in this thesis:

- We write $A, B \implies A \wedge B$ instead of $\{A, B\} \implies \{A \wedge B\}$
- To indicate that some specific formulas are part of an arbitrary antecedent (or succedent), we write $\Gamma, A \implies \Delta, A, B$ instead of $\Gamma \cup \{A\} \implies \Delta \cup \{A, B\}$

Theorem 1.

$$S \coloneqq (\Gamma, \Delta) \textit{ is invalid iff } \varphi(S) \coloneqq \left(\bigwedge_{\psi \in \Gamma} \psi \right) \wedge \left(\bigwedge_{\psi' \in \Delta} \neg \psi' \right) \textit{ is satisfiable.}$$

Proof. ($\Leftarrow$) If $\varphi(S)$ is satisfiable, an interpretation I must exist that satisfies $\varphi(S)$. This interpretation satisfies all formulas $a \in \Gamma$ but none in Δ and thus the sequent is invalid. If $\varphi(S)$ is unsatisfiable, no such interpretation exists and because of that the sequent is valid. The argumentation for the forward implication ($\Rightarrow$) works in the same way. $\square$

3.1 The propositional Set of Rules

The Sequent Calculus defines a set of rules with reference to the validity definition of sequents. Every rule is made out of premises and a conclusion where the conclusion is logically derived from the premises.

$$\frac{\text{premise}}{\text{conclusion}}\,(\text{rule name})$$

To get an idea of why a rule is sound and correct, it is necessary to understand the definition of the validity of sequents. The antecedent is satisfied by a set of interpretations ($\mathcal{I}$) and all these interpretations must satisfy one formula out of the succedent. If the antecedent or the succedent changes, the set $\mathcal{I}$ may change or some of these interpretations do not satisfy a formula in the succedent anymore. As a consequence, it is necessary to check if a rule preserves this implication: "*all interpretations* $I \models \Gamma$ *must at least satisfy one formula in* Δ*"*.

The premise may contain multiple sequents that are needed to make the conclusion. The conclusion is only proved to be valid, if all sequents from the premise are proved to be valid. Rules with an empty premise are called *axiom*. The conclusion of an axiom is thus valid by definition. In other words, the conclusion of an axiom contains a sequent that is valid in a trivial way.

$$\frac{}{\Gamma, \varphi \implies \Delta, \varphi}\,(Ax)$$

If a formula φ is part of the antecedent, then all interpretations that satisfy the antecedent must satisfy φ. And if φ is also part of the succedent, then all of these interpretations satisfy at least one formula out of the succedent: φ. Hence, the sequent must be valid.

$$\frac{}{\Gamma, \mathrm{ff} \implies \Delta}\,(\mathrm{ff}_L)$$

If the false constant is part of the antecedent, then no interpretation exists that satisfies all formulas in the antecedent because the formula ff is unsatisfiable. As a result, if no interpretation satisfies the antecedent, the implication of the validity definition for sequents is true in a trivial way.

$$\frac{}{\Gamma \implies \Delta, \mathrm{tt}}\,(\mathrm{tt}_R)$$

If the true constant is part of the succedent, then all interpretations are able to satisfy

at least one formula of the succedent: the (valid) true constant. Thus, the sequent must be valid.

In order to reduce more complex sequents into axioms, the Sequent Calculus offers rules for every first-order operator. The definition for each rule starts from a set of sequents that are expected to be valid – the premise – and derives a conclusion from these sequents. This is the top-down perspective and it is only helpful in reading finished proofs. But when you want to prove a given sequent, you create the proof tree the other way round. Starting from the resulting sequent as a conclusion, premises are built out of conclusions. This is the bottom-up perspective. To offer a short notion and legitimization for each rule we may switch between the two perspectives.

$$\frac{\Gamma, \varphi, \psi \implies \Delta}{\Gamma, \varphi \wedge \psi \implies \Delta} (\wedge_L)$$

The conjunctive definition of the antecedent allows this form of syntactic transformation. An interpretation that satisfies the antecedent must be model of Γ *and* φ *and* ψ in both sequents, which means that the semantic does not change. Hence, if the premise is valid, the conclusion must also be valid.

$$\frac{\Gamma \implies \Delta, \varphi \qquad \Gamma \implies \Delta, \psi}{\Gamma \implies \Delta, \varphi \wedge \psi} (\wedge_R)$$

The conclusion contains the formula $\varphi \wedge \psi$. To prove its validity we can split this sequent into two subsequents that only differ in φ and ψ. If both subsequents are valid, we can conclude the following: all interpretations $I \models \Gamma$ satisfy either a formula in Δ or I is model of φ and ψ together. Otherwise, one of the two subsequents should have been invalid. And if these interpretations satisfy φ and ψ together, they also satisfy $\varphi \wedge \psi$.

$$\frac{\Gamma, \varphi \implies \Delta \qquad \Gamma, \psi \implies \Delta}{\Gamma, \varphi \vee \psi \implies \Delta} (\vee_L)$$

An antecedent $A := \Gamma \cup \{\varphi \vee \psi\}$ requires all interpretations $I \models A$ to satisfy φ or ψ. Thus, the set $\mathcal{I}$ of all interpretations $I \models A$ is split into two subsets: $\mathcal{I} = \mathcal{I}_\varphi \cup \mathcal{I}_\psi$, the set of interpretations $\mathcal{I}_\varphi$ that are model of φ and those interpretations $\mathcal{I}_\psi$ that are model of ψ. One can easily see that each sequent in the premise covers exactly one of these subsets of interpretations. From the validity of the two subsequents we can conclude that all interpretations in $\mathcal{I}_\varphi$ and $\mathcal{I}_\psi$ satisfy at least one formula in Δ. Hence, all interpretations in $\mathcal{I}$ satisfy at least one formula in Δ. This is exactly the definition for the validity of the conclusion.

$$\frac{\Gamma \implies \Delta, \varphi, \psi}{\Gamma \implies \Delta, \varphi \vee \psi} (\vee_R)$$

Because of the disjunctive definition of the succedent this rule is nothing but a syntactical transformation with no effect on the semantics. All interpretations $I \models \Gamma$ must either satisfy a formula in Δ or φ or ψ. This semantic holds for both sequents.

$$\frac{\Gamma \implies \Delta, \varphi}{\Gamma, \neg\varphi \implies \Delta}\,(\neg_L)$$

Starting from the conclusion, if $\neg\varphi$ is deleted from the antecedent, the set of interpretations that satisfy the antecedent is extended by interpretations that also satisfy φ. These interpretations may not satisfy any formula in the succedent. To recover the validity of the sequent φ is included in the succedent.

$$\frac{\Gamma, \varphi \implies \Delta}{\Gamma \implies \Delta, \neg\varphi}\,(\neg_R)$$

Starting again from the conclusion, if $\neg\varphi$ is deleted from the succedent, the sequent may become invalid because some interpretations $I \models \Gamma$ satisfy $\neg\varphi$ and nothing else in Δ. Exactly these interpretations are excluded by adding φ to the antecedent.

$$\frac{\Gamma \implies \Delta, \varphi \qquad \Gamma, \psi \implies \Delta}{\Gamma, \varphi \to \psi \implies \Delta}\,(\to_L)$$

Implications like $\varphi \to \psi$ are equivalent to $\neg\varphi \vee \psi$. This rule is a shorthand that combines the $\vee_L$-rule and the $\neg_L$-rule.

$$\frac{\Gamma, \varphi \implies \Delta, \psi}{\Gamma \implies \Delta, \varphi \to \psi}\,(\to_R)$$

This rule is again a shorthand but this time for the $\vee_R$-rule followed by the $\neg_R$-rule.

$$\frac{\Gamma \implies \Delta, \varphi, \psi \qquad \Gamma, \varphi, \psi \implies \Delta}{\Gamma, \varphi \leftrightarrow \psi \implies \Delta}\,(\leftrightarrow_L)$$

A bi-implication like $\varphi \leftrightarrow \psi$ is among others equivalent to $(\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi)$ (both or none at all). The $\leftrightarrow_L$-rule acts as a shorthand for the $\vee_L$-rule followed by two applications of the $\wedge_L$-rule and finally two times the $\neg_L$-rule.

$$\frac{\Gamma, \psi \implies \Delta, \varphi \qquad \Gamma, \varphi \implies \Delta, \psi}{\Gamma \implies \Delta, \varphi \leftrightarrow \psi}\,(\leftrightarrow_R)$$

Bi-implications like $\varphi \leftrightarrow \psi$ are *also* equivalent to $(\varphi \to \psi) \wedge (\psi \to \varphi)$ (both formulas imply each other). The $\leftrightarrow_L$-rule acts as shorthand for the $\wedge_R$-rule followed by two applications of the $\to_R$-rule.

An important property of all rules that are shown so far is that they reduce the number of operators used per sequent. Because of the finite number of operators per formula every path in a sequent tree must end with a sequent where no more rule applications are possible. Moreover, the validity of each premise implies the validity of the conclusion and the other way round. Hence, it is not possible to create invalid sequents from a valid one by applying rules in a wrong order. This is why we are able to apply rules in an arbitrary order to find a complete proof. If an invalid sequent is reached by using these rules, then all sequents in the path down to the root sequent must be invalid as well.

It is even possible to create an algorithm that decides the validity of a sequent as long as it is limited to the quantifier-free set of operators: $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ (see Algorithm 1). Equalities are also disallowed so far but we will cover this issue in detail in Section 4.1.

Algorithm 1 Algorithm to decide the validity of sequents over $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ without equality

```
1:  procedure isValid(s_R : Sequent): Boolean
2:      var todo := [s_R];
3:      while todo is not empty do
4:          var s := get and remove first sequent from todo
5:          if a rule application on s is possible then
6:              var r := pick applicable rule for s
7:              var newSequents := apply rule r on s
8:              todo := todo + newSequents
9:          else
10:             return false;
11:         end if
12:     end while
13:     return true
14: end procedure
```

Legend

:= is used as the assignment operator

[ ] is used for lists

: is used to define types

\+ is used to concatenate lists

To handle equalities and quantifiers we need some more rules, which are part of the subsequent section. We will see in Section 4.1 that the subset $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \doteq\}$ is still decidable by creating a congruence closure over the set of equalities. The validity problem for sequents becomes undecidable when quantifiers are added. Nevertheless, the whole Sequent Calculus for FO[=] is sound [EFT07, chapter 4] and complete [EFT07, chapter 5].

3.2 Rules for Equalities and Quantifiers

The dotted equality is a predefined relation with the semantics of the natural equality. Because it is predefined, we have to handle it with respect to its definition, especially for the reflexivity and congruence property.

$$\frac{}{\Gamma \implies \Delta, t \doteq t}\,(Refl_R)$$

The dotted equality relation inherits the reflexivity property from the known equality relation. Hence, statements like $t \doteq t$ are always true by definition. The argumentation for the validity of the sequent follows the one of the tt_R-rule.

$$\frac{\Gamma, t \doteq t \implies \Delta}{\Gamma \implies \Delta}\,(Refl_L)$$

Because every interpretation is a model of $t \doteq t$, where t is a ground term, it is always possible to add such formulas to the antecedent without changing the set of interpretations that satisfy the antecedent.

$$\frac{\Gamma,\ t_1 \doteq t_2,\ \varphi[t_2]_p \implies \Delta}{\Gamma,\ t_1 \doteq t_2,\ \varphi[t_1]_p \implies \Delta}\,(Subst_L) \qquad \frac{\Gamma,\ t_1 \doteq t_2 \implies \Delta,\ \varphi[t_2]_p}{\Gamma,\ t_1 \doteq t_2 \implies \Delta,\ \varphi[t_1]_p}\,(Subst_R)$$

If the antecedent contains an equality like $t_1 \doteq t_2$, all interpretations must satisfy this formula. This is why we can assume the two terms to be equal. But in order to apply the $Relf_R$-rule or the Ax-rule we need to make formulas *syntactically* equal. Because of the given semantics it is possible without loss of generality to replace the terms by each other. In other words, the term t_1 at position p in a formula φ can be replaced by the term t_2.
Equalities in the succedent cannot be used to legitimate a substitution step. To get an idea about the reason one can negate an equality on the right side twice, which is a no-op, and use the $\neg_R$-rule afterwards. As a result, the equality appears in a single negated version on the left side. Therefore, we can assume both terms of the equality to be *unequal*. Because we suppose all ground terms to be different by definition this is not a new information.

An equality $t_1 \doteq t_2$ allows substitutions in both directions: t_1 can be replaced by t_2 and vice versa. This fact makes it possible, in particular, to change the terms back and forth as often as desired. Among other things this is the reason why in general arbitrary rule applications do not lead to success anymore when a sequent contains equalities. The same holds for rules used to handle quantifiers.

While using the substitution rules, it is sometimes necessary to have both instances of a formula at hand – the one before the substitution rule has been applied and the one after the application. For this purpose we introduce a duplication rule (bottom-up perspective) that is called the contraction rule (top-down perspective).

$$\frac{\Gamma,\ \varphi, \varphi \implies \Delta}{\Gamma,\ \varphi \implies \Delta}\,(Contr_L) \qquad \frac{\Gamma \implies \Delta,\ \varphi, \varphi}{\Gamma \implies \Delta,\ \varphi}\,(Contr_R)$$

Adding the same formula to the antecedent or succedent that is already contained does not change anything. But it is useful to make a copy of a formula before it is modified by a rule. Sets normally do not contain the same elements twice, but when you think

of it as an instance based set this is fine. These rules are also needed in connection with quantifiers. Beside the contraction rules we need four additional rules to handle quantifiers – one for each quantifier and each side.

$$\frac{\Gamma, \varphi[c/x] \implies \Delta}{\Gamma, \exists x\, \varphi \implies \Delta}\,(\exists_L)$$

As always, we have to take all interpretations into account that satisfy the antecedent and all these interpretations using different definitions for relation and function symbols. This is why the element used to satisfy $\exists x\, \varphi$ may differ from interpretation to interpretation. As a result we cannot pick only one or even a finite set of ground terms for the instantiation. But it is possible to avoid the choice of one specific ground term by using a new constant. With such a new constant every interpretation must be extended by a definition for this constant. Among all new interpretations that extend the previous interpretation $I \models \exists x\, \varphi$ there is the one that defines the new constant to be exactly that element used to satisfy $\exists x\, \varphi$ under I. Thus, for each previous interpretation at least one interpretation exists that satisfies the antecedent when we use the new constant for instantiation. Interpretations that define a wrong definition for the new constant do not satisfy the antecedent and are thus excluded. This in mind, we can use the new constant to instantiate φ without changing the semantics. The new constant acts like an existential quantification in the antecedent.

$$\frac{\Gamma \implies \Delta, \varphi[c/x]}{\Gamma \implies \Delta, \forall x\, \varphi}\,(\forall_R)$$

The rule for universal quantifiers in the succedent behaves in the same way as the $\exists_L$-rule. For the set of interpretations $\mathcal{I}$, where $I \in \mathcal{I}$ satisfies Γ and $\forall x\, \varphi$ – and perhaps nothing else in the succedent –, we create a new set $\mathcal{I}'$ by adding a new constant. The new set contains one interpretation for each previous interpretation multiplied by *all possible assignments* for the new constant. The $\exists_L$-rule also adds a new constant but it modifies the antecedent so that only some new interpretations satisfy the antecedent. This time the antecedent does not change (Γ), so all new interpretations satisfy the antecedent and we have to consider all of them in order to validate the sequent. Certainly, this is just the same semantic as for a universal quantifier. In other words, the set of new interpretations that are created by adding a new constant and that cover all possible assignment for the new constant allows us to use the constant as a universal quantification in the succedent.

$$\frac{\Gamma,\ \varphi[t/x] \implies \Delta}{\Gamma,\ \forall x\, \varphi \implies \Delta}\,(\forall_L)$$

The $\forall_L$-rule instantiates all occurrences of the bound variable x in φ with a specific ground term. If $\mathcal{I}_\mathcal{C}$ is the set of interpretations that satisfy the antecedent of the conclusion and $\mathcal{I}_\mathcal{P}$ the set of interpretations that satisfy the antecedent of the premise, then $\mathcal{I}_\mathcal{C} \subset \mathcal{I}_\mathcal{P}$. The reason for this is that we reduce the statement "φ *is true for all ground terms*" to "φ *is true for the ground term* t". This allows a lot more interpretations to satisfy the antecedent. If the premise is valid, all interpretations $I \in \mathcal{I}_\mathcal{P}$ satisfy at least

one formula in Δ. In this case all interpretations $I \in \mathcal{I}_{\mathcal{C}}$ must also satisfy at least one formula in Δ because $\mathcal{I}_{\mathcal{C}} \subset \mathcal{I}_{\mathcal{P}}$.

It is important to notice that we cannot inverse the rule, which means that the validity of the premise cannot be derived from the validity of the conclusion. In contrast to all rules shown so far, the premise can be invalid while the conclusion is valid. In other words, if you have picked the wrong ground term, the premise becomes invalid even if the conclusion is valid.

$$\frac{\Gamma \implies \Delta, \varphi[t/x]}{\Gamma \implies \Delta, \exists x\, \varphi}\,(\exists_R)$$

Similarly to the $\forall_L$-rule we must pick a ground term for the instantiation. If an interpretation I satisfies the antecedent and $\exists x\, \varphi$ but none in Δ, we must ensure that I is also a model of $\varphi[t/x]$ by choosing a ground term that satisfies φ under the interpretation I. Choosing the wrong ground term results in an invalid sequent. Maybe it is the case that two different interpretations need different ground terms to satisfy φ. This requires to make multiple instantiations. The requirement of multiple instantiations applies to the $\forall_L$-rule *and* the $\exists_R$-rule. See, for example, the following two sequents:

$$\forall x.\, P(x) \land \neg(P(c) \land P(d)) \implies \emptyset \tag{1}$$
$$\emptyset \implies \exists x.\, P(x) \lor \neg(P(c) \lor P(d)) \tag{2}$$

One possible way to prove the first sequent is the following sequent tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(c),\ P(d) \implies P(d),\ P(c) \land P(d)}\,(Ax) \quad \dfrac{}{P(c),\ P(d) \implies P(c),\ P(c) \land P(d)}\,(Ax)}{P(c),\ P(d) \implies P(c) \land P(d),\ P(c) \land P(d)}\,(\land_R)}{P(c),\ P(d),\ \neg(P(c) \land P(d)) \implies P(c) \land P(d)}\,(\neg_L)}{P(c),\ P(d) \land \neg(P(c) \land P(d)) \implies P(c) \land P(d)}\,(\land_L)}{P(c),\ \neg(P(c) \land P(d)),\ P(d) \land \neg(P(c) \land P(d)) \implies \emptyset}\,(\neg_L)}{P(c) \land \neg(P(c) \land P(d)),\ P(d) \land \neg(P(c) \land P(d)) \implies \emptyset}\,(\land_L)}{P(c) \land \neg(P(c) \land P(d)),\ \forall x.\, P(x) \land \neg(P(c) \land P(d)) \implies \emptyset}\,(\forall_L)}{\forall x.\, P(x) \land \neg(P(c) \land P(d)),\ \forall x.\, P(x) \land \neg(P(c) \land P(d)) \implies \emptyset}\,(\forall_L)}{\forall x.\, P(x) \land \neg(P(c) \land P(d)) \implies \emptyset}\,(Contr_L)$$

While the rule $Refl_L$ is somehow useful to create short proofs, it is not necessary for the Sequent Calculus to be complete. This is the case because the $Refl_L$-rule serves only one use case: adding a context to known equalities out of the antecedent. The resulting equality can then be used for other substitutions. Nevertheless, it is always possible to avoid the context creation step by doing the substitution within the target term (in place).

Theorem 2. *The rule $Refl_L$ can be removed from the set of available rules without loss of completeness for the Sequent Calculus.*

Proof. Let S_1 be the *last* sequent in a path from the root sequent up to an axiom within a sequent tree where the $Refl_L$ is applied to. Without loss of generality we assume that all operators are already resolved at this point and all subsequent rule applications in the path are based on substitutions, the $Refl_R$-rule and the Ax-rule. This assumption is possible due to the fact that the $Refl_L$ is not required to resolve any operator. Because all remaining rules have just one or no premise the sequent tree does not branch out any more.
The proof for the hypothesis is now made by an induction over the length l of the path that starts at the sequent S_1 and ends at the last sequent S_l. The length of a path is given by the number of sequents that are elements of this path. With the assumption that the $Refl_L$ is applied to the sequent S_1 the path must have at least a length of two. Note that such a path always starts with an application of the $Refl_L$-rule, ends with one of the two axioms and is made only out of substitutions in between. Let $t \doteq t$ be the equality that has been added by the application of the $Refl_L$-rule.

If $l = 2$, then two rule applications are made. The first one is the $Refl_L$-rule and the second one must be an axiom. The only case where the equality $t = t$ is involved in the application of the axiom is when the Ax-rule is used in combination with this equality. In that case $t \doteq t$ must also be part of the succedent which allows to use the $Refl_R$-rule instead. Therefore, the equality $t \doteq t$ is not needed in its original form to prove the sequent and is not modified or used by any substitution step – because there is no substitution step.

Induction step $l = n + 1$: Compared to all paths with the length n a path of length $n + 1$ contains just one more substitution step. From the induction hypothesis results that all other substitution steps do not use $t \doteq t$ or modified versions of it. If the new substitution step does not modify $t \doteq t$ either, the argumentation follows the one from the induction basis to prove that we do not need the rule $Refl_L$. If it modifies $t \doteq t$, we are able to do the following modification. Consider $t' \doteq t$ to be the modified version of $t \doteq t$ that is created by the substitution step $t' = t[u/v]_p$. The new equality $t' \doteq t$ is only important for the proof of the sequent if it is used by the Ax-rule in the last step. This implies that the equality $t' \doteq t$ is also part of the succedent. In this case the substitution step can be inverted and used to modify $t' \doteq t$ in the succedent ($t = t'[v/u]_p$) instead of $t \doteq t$ in the antecedent. The result is a sequent that contains $t \doteq t$ in the succedent, which is why the $Refl_R$-rule is applicable. With this modification we are able to conclude for paths with the length $n+1$ that the equality $t \doteq t$ is not needed in the original form to prove the sequent and is not modified or used by any substitution step. □

4 Proof Search

A common way to solve problems in computer science is to try all possible solutions one after another until the right solution is found (exhaustive search). As shown in Algorithm 1 this is quite easy for sequents over the propositional set of operators $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ as it is sufficient to try only one applicable combination of rules – a search space of one. With a simple equality like $c \doteq f(c)$ the number of possible substitution steps is already infinite – the substitution where c is replaced by $f(c)$ could by used infinitely often. With quantifiers, even more decisions have to be made that dramatically increase the search area. How many instances do you need from a single quantified formula and which ground term should be used for each instance? Summed up, we have infinitely many possibilities for duplications per quantified formula, infinitely many possibilities to instantiate each quantified formula and again infinitely many possibilities to do substitutions.[I]
To do an exhaustive search for the whole Sequent Calculus over FO[=] we have to ensure that all solutions are reached in a finite number of steps. The problem is that we cannot do it by a depth-first search over duplications, instantiations and substitutions. E.g. choose a number of duplications for each formula and subformula, choose a matching set of ground term instantiations, choose a finite set of substitutions steps and check the resulting sequents for validity using Algorithm 1. If it is invalid, change the last decision (backtracking). A depth-first search like this would always be stuck in an infinite path. As a result a breadth-first search is needed.

One possible breadth-first search algorithm may follow the steps shown in Algorithm 2. A term enumerator is used to instantiate quantified formulas which is a stateful function that is defined over an ordered list of ground terms. It accepts a formula as an argument and returns the first term out of the ordered list of ground terms that the term enumerator has not already returned for the given formula so far.
Every time when multiple rule applications are possible for a formula φ the algorithm makes a copy of φ to try all possibilities because we do not know which one is right. The second important thing is how the queue is handled. Adding modified formulas to the end of the queue ensures that all formulas in a queue are processed after a finite number of steps (fairness of formulas). Adding new or modified queues to the end of the queue-list ($\hat{Q}$) ensures that all sequents will be processed over time (fairness of queues). The problem of this naive algorithm is that the number and sizes of queues quickly explodes even for small formulas. Take, for example, the following simple sequent:

$$\forall x\ .P(x) \vee Q(x), \neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg A \implies A$$

The naive algorithm stated above would create thousands of copies for the quantified formula until the second formula is reduced to A because of its fairness constraints. Each instance of $P(\hat{x}) \vee Q(\hat{x})$ forces the algorithm to fork the current queue. Every fork creates new copies of the quantified formula and gets forked as well, which results in

[I] Well, that is three times infinity and even Chuck Norris can only count to infinity twice.

Algorithm 2 Algorithm to semi-decide the validity of sequents for FO[=]

1. All formulas from the root sequent are enqueued into a first-in-first-out (FIFO) queue $Q_{R,TE}$ that is associated to the root sequent R and a new term enumerator TE. The queue itself is enqueued into a FIFO queue of queues $\hat{Q}$.

2. If one of the axiom rules can be applied to a leaf sequent, the sequent and its queue are dropped. If no sequent is left, the algorithm returns true.

3. Pick the first queue $Q_{S,TE}$ out of all queues $\hat{Q}$, which is associated to the sequent S and the term enumerator TE, and pick the first formula φ out of the queue $Q_{S,TE}$. Follow all these steps:

   a) For every possible substitution step for φ create a copy of φ using the contraction rule and apply the substitution step to it. Add φ and its modified versions to the end of $Q_{S,TE}$.

   b) If one of the rules $\{\neg_L, \neg_R, \wedge_L, \vee_R, \rightarrow_R, \exists_L, \forall_R\}$ is applicable to φ, apply it and add the new formula to the end of $Q_{S,TE}$. Add $Q_{S,TE}$ to the end of $\hat{Q}$.

   If one of the rules $\{\exists_R, \forall_L\}$ is applicable to φ, create a copy using the contraction rule and apply the rule using the ground term $t = TE(\varphi)$. Add φ and its modified version to the end of $Q_{S,TE}$. Add $Q_{S,TE}$ to the end of $\hat{Q}$.

   If one of the rules $\{\vee_L, \wedge_R, \rightarrow_L, \leftrightarrow_L, \leftrightarrow_R\}$ is applicable to φ apply it and create a new queue for each subsequent: $Q_{S',TE'}$ and $Q_{S'',TE''}$, where TE' and TE'' are copies of TE. Add the new formulas to the end of the corresponding queue and add both queues to the end of $\hat{Q}$.

   c) Repeat step 2.

an exponentially growing number of queues. If the sequent contained more quantified formulas it would grow even faster.

Algorithm 2 could easily be modified to return a finished sequent tree instead of just *true*. But as outlined in Chapter 1, we need an algorithm that is fast and human-readable in order to make the Sequent Calculus Trainer smart and Algorithm 2 is neither fast nor does it produce human-readable proofs. Additionally, it needs an exponential amount of space.

4.1 Solving Equalities

The Sequent Calculus has four rules to handle the predefined dotted equality ($Refl_L$, $Refl_R$, $Subst_L$, $Subst_R$) and as shown in Theorem 2 the $Refl_L$-rule can be replaced in favour of the substitution rules without the loss of completeness. One important

aspect of the substitution rules is that they can only be used to replace the term t_1 by t_2 if the equality $t_1 \doteq t_2$ is an element of the antecedent. E.g. if the formula $A \vee t_1 \doteq t_2$ is an element of the antecedent we cannot assume t_1 and t_2 to be equal, because not all interpretations have to satisfy the right part of the disjunction. Another example, when an equality is bound by a quantifier ($\forall x\ x \doteq f(c)$) it is even unclear which equality statement results from the instantiation of the quantified formula ($c \doteq f(c)$, $f(f(c)) \doteq f(c)$). Hence, a good strategy to handle equalities is to eliminate all the other operators first and do the equality check afterwards. What is left after the operator elimination is done is a set of sequents that only contain atomic propositions (including equalities). The resulting question is: can the set of equalities within the antecedent be used to make either two formulas – one in the antecedent and one in the succedent – syntactically equal or can it be used to create an equality in the succedent that is made out of two syntactically equal terms. In the first case the axiom rule can be used to complete the sequent tree and in the second case the $Refl_R$-rule is applicable. In both situations it is necessary to make a pair of terms syntactically equal by using the available set of equalities in the antecedent. E.g. to make $R(c, f(d))$ syntactically equal to $R(d, e)$ we have to show that c is equal to d and $f(d)$ is equal to e using the substitution rules. The same applies to an equality $c \doteq d$ on the right side, where we have to show that c and d are equal using equalities from the left side. So the above question can be condensed to:

The equality creation problem. If $\Gamma \Longrightarrow \Delta$ is a sequent, $E_\Gamma \subseteq \Gamma$ is the set of equalities in the antecedent (Γ) and s, t are some terms out of the given sequent, is it possible to apply contraction and substitution rules such that a new sequent $\Gamma' \Longrightarrow \Delta'$ is created with $s \doteq t \in \Gamma'$.
Note that the contraction and substitution rules are both made from a single premise. So the application of such a rule to a sequent always results in a new modified version of this sequent. The next three examples demonstrate the equality creation problem:

$$c \doteq d,\ d \doteq e \implies c \doteq e \tag{1}$$

$$c \doteq d,\ d \doteq e,\ P(c) \implies P(e) \tag{2}$$

$$c \doteq d,\ d \doteq e,\ P(f(c)) \implies P(e) \tag{3}$$

$$c \doteq f(f(c)),\ d \doteq f(f(f(d))),\ f(c) \doteq e,\ f(d) \doteq e \implies c \doteq d \tag{4}$$

For the sequent 1 the set of left sided equalities is $E_\Gamma = \{c \doteq d,\ d \doteq e\}$. To prove the sequent we need to show that the equality statement $c \doteq e$ follows from the succedent. So according to the equality creation problem we can ask if a list of contraction and substitution applications can be used to create a new sequent with a left sided equality $c \doteq e$ (or $e \doteq c$). The same holds for the sequent 2. Sequent 3, in contrast, is invalid and thus we expect a negative solution for the equality creation problem with the equality $f(c) \doteq e$. While it is easy to see for sequent 1 and 2 which steps are needed to solve the corresponding equality creation problem, it is not the case for the last more complex example. We need at least six substitution applications for a solution of the equality creations problem given by sequent 4:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{c \doteq d,\ d \doteq f(d),\ f(c) \doteq f(d),\ f(d) \doteq e \implies c \doteq d}(Ax)}{c \doteq f(d),\ d \doteq f(d),\ f(c) \doteq f(d),\ f(d) \doteq e \implies c \doteq d}(Subst_L)}{c \doteq f(d),\ d \doteq f(c),\ f(c) \doteq f(d),\ f(d) \doteq e \implies c \doteq d}(Subst_L)}{c \doteq f(f(c)),\ d \doteq f(c),\ f(c) \doteq f(d),\ f(d) \doteq e \implies c \doteq d}(Subst_L)}{c \doteq f(f(c)),\ d \doteq f(f(c)),\ f(c) \doteq f(d),\ f(d) \doteq e \implies c \doteq d}(Subst_L)}{c \doteq f(f(c)),\ d \doteq f(f(f(d))),\ f(c) \doteq f(d),\ f(d) \doteq e \implies c \doteq d}(Subst_L)}{c \doteq f(f(c)),\ d \doteq f(f(f(d))),\ f(c) \doteq e,\ f(d) \doteq e \implies c \doteq d}(Subst_L)$$

A similar problem exists in the study of *term rewriting systems* [BN98]: *the word problem of ground identities*. We will show that the word problem of ground identities is decidable and that the equality creation problem can be reduced to the word problem. While the prove is based on [BN98, Section 4.3] and in particular [BN98, Theorem 4.3.5] we will adjust it to fit the notation of this thesis and combine it with the context of the Sequent Calculus. Considering the results of [BN98, Section 3.1], we then modify the decision procedure for the word problem so that the equality creation problem can be solved directly without the reduction step. The last part of this section is about a short algorithm that can be used to decide the validity of quantifier-free sequents in combination with the decision procedure for the equality creation problem.

In order to make a formal definition of the word problem of ground identities we need the notion of congruence closures first. From now on let Σ be the set of available function symbols, G_Σ be the set of all ground terms over Σ and E be a finite set of term pairs (ground identities).

Definition 12. *The congruence closure $CC(E)$ for a given set of equalities E is the least set that contains E and is closed under reflexivity, symmetry, transitivity and congruence.*

With the subsequent definition a set is a congruence closure iff it is closed under *Cong*:

$$\begin{aligned}
Re(\hat{E}) &\coloneqq \{(u,u) \mid u \in G_\Sigma\} \\
Sy(\hat{E}) &\coloneqq \{(u,v) \mid (v,u) \in \hat{E}\ \} \\
Tr(\hat{E}) &\coloneqq \{(u,w) \mid \exists v.\ (u,v),(v,w) \in \hat{E}\} \\
Co(\hat{E}) &\coloneqq \{(f(u_1,\ldots,u_n), f(v_1,\ldots,v_n)) \mid f^{(n)} \in \Sigma \wedge (u_1,v_1),\ldots,(u_n,v_n) \in \hat{E}\} \\
Cong(\hat{E}) &\coloneqq \hat{E} \cup R(\hat{E}) \cup S(\hat{E}) \cup T(\hat{E}) \cup C(\hat{E})
\end{aligned}$$

The one-step congruence closure $Cong(\hat{E})$ accepts a set of term pairs $\hat{E}$ and calculates separately the reflexive, symmetrical, transitive and congruent pairs for $\hat{E}$. As shown in [BN98, p. 62-63] the process of closing E under *Cong* is an iteration using the

following sequence:

$$C_0 \coloneqq E$$
$$C_{i+1} \coloneqq Cong(C_i)$$

With this sequence we can define the congruence closure $CC(E)$ for a set E as

$$CC(E) \coloneqq \bigcup_{i \geq 0} C_i$$

The word problem of ground identities. Given a set of term pairs E and a single term pair (s, t): is $(s, t) \in CC(E)$?

If at least one function symbol exists that has an arity of one or greater, then such a congruence closure is an infinitely large set because of its congruence property. So it is impossible to decide whether a pair of terms is an element of the congruence closure by calculating *all* C_i iteratively. But as outlined in [BN98], the search space for the word problem is finite. We only have to consider all terms and subterms in $E \cup \{s, t\}$. For this purpose we define the finite set T for the word problem as the least set that contains $\{u \mid (u, v) \in E \vee (v, u) \in E\} \cup \{s, t\}$ and is additionally closed under subterms: if $t_s \Subset_p t$ and $t \in T$, then $t_s \in T$.

Definition 13. *$CC_T(E) \coloneqq CC(E) \cap (T \times T)$ is the congruence closure that is restricted by the set of terms T.*

Similarly to the sequence C we define the sequence D, which accumulates the results of multiple *Cong* executions with the restriction to the set of terms T.

$$D_0 \coloneqq E$$
$$D_{i+1} \coloneqq Cong(D_i) \cap T \times T$$

Because of the T-limitation and its monotonicity this sequence must converge after a finite number of steps. $T \times T$ is finite and $\bigcup_{i \geq 0} D_i$ cannot contain more elements than $T \times T$. For this reason there must be a point m in the sequence D such that $D_m = D_{m+1}$. With this in mind, D_m is now used to talk about the stable point of the sequence D. Furthermore, elements are never dropped from one sequence step to another (monotonicity) and thus $D_i \subseteq D_{i+1}$.
The difference between D_m and $CC_T(E)$ is the point where the restriction to the elements in $T \times T$ applies. For the sequence D this is done for every single step ($\forall i \; D_i \subseteq T \times T$). In contrast, $CC_T(E)$ is defined as the full congruence closure $CC(E)$ which is then reduced to the elements that are also contained in $T \times T$.

The equivalence of both definitions is element of the next two theorems. Because transitivity makes it complicated to argue about chains of terms we define a variation of CC_T first. This variation is then used to prove $D_m = CC_T(E)$.

Definition 14. *CC_T^- is a variation of the congruence closure CC_T. It has the same*

definition as CC_T except for the transitivity property, which is omitted.

Definition 15. *We say (u, v) is transitive in $CC_T^-(E)$ iff it is possible to create a sequence of terms $w_1, w_2, \ldots, w_n$ (called a transitive chain) that starts in $u = w_1$, ends in $v = w_n$ and for which each pair (w_i, w_{i+1}) $(0 \leq i < n)$ in the sequence is also a pair in $CC_T^-(E)$: $(w_i, w_{i+1}) \in CC_T^-(E)$.*

Theorem 3. *For an arbitrary pair (u, v) the following holds: $(u, v) \in CC_T(E)$ iff (u, v) is transitive in $CC_T^-(E)$*

Proof. The transitive chain of term pairs is nothing else but a transitivity closure on top of $CC_T^-(E)$. If a pair (w_1, w_n) is transitive in $CC_T^-(E)$, it must also be an element of $CC_T(E)$ because $CC_T^-(E) \subseteq CC_T(E)$ and the transitivity property of $CC_T(E)$ allows to conclude $(w_1, w_n) \in CC_T(E)$.
The other direction can be proved by an induction over the sequence C. Note that for every pair $(u, v) \in CC_T(E)$ there has to be an iteration i of the sequence C such that $(u, v) \in C_i \cap T \times T$.

Induction base (C_0): $C_0 = E \subseteq CC_T^-(E)$ and thus the sequence of w_1 followed by w_2 with $(w_1, w_2) \in C_0 \subseteq CC_T^-(E)$ is always a valid transitive chain over $CC_T^-(E)$.

Induction step (C_{i+1}): We can conclude from the induction hypothesis, that each pair $(u, v) \in C_i \cap T \times T$ is also transitive in $CC_T^-(E)$. For every new pair in C_{i+1} the following holds:

For every pair $(w, w) \in C_{i+1} \cap T \times T$ we have that (w, w) is a valid transitive chain over $CC_T^-(E)$ because of its reflexivity property.

If $(w_n, w_1) \in C_{i+1}$ and $(w_1, w_n) \in C_i \cap T \times T$, then a transitive chain $(w_1, \ldots, w_n)$ must exist such that (w_1, w_n) is transitive in $CC_T^-(E)$. Definition 15 says that every pair (w_j, w_{j+1}) $(0 \leq j < n)$ of such a chain is an element of $CC_T^-(E)$ and from the symmetry property of $CC_T^-(E)$ results that $(w_n, \ldots, w_1)$ is also a valid transitive chain over $CC_T^-(E)$.

If $(w_1, w_n) \in C_{i+1}$ and $(w_1, w_m), (w_m, w_n) \in C_i \cap T \times T$, then $(w_1, \ldots, w_n)$ is a valid transitive chain over $CC_T^-(E)$ that is built by the concatenation of the chains $(w_1, \ldots, w_m)$ and $(w_m, \ldots, w_n)$.

If $f^{(n)} \in \Sigma$, $(f(w_{1,1}, \ldots, w_{n,1}), f(w_{1,m}, \ldots, w_{n,m})) \in C_{i+1} \cap T \times T$ and $(w_{j,1}, w_{j,m}) \in C_i \cap T \times T$ with $1 \leq j \leq n$, then we can conclude from the induction hypothesis to have a transitive chain $(w_{j,1}, \ldots, w_{j,m})$ for every j with $1 \leq j \leq n$. Theses chains may vary in their length but we assume all chains to have the same length because shorter chains can be extended by concatenating (u, u) to the end of the chain if u was the last element. Every pair $(w_{j,k}, w_{j,k+1})$ $(1 \leq j \leq n,\ 1 \leq k < m)$ is an element of $CC_T^-(E)$ and the congruence property of $CC_T^-(E)$ allows to conclude that $(f(w_{1,k}, \ldots, w_{n,k}), f(w_{1,k+1}, \ldots, w_{n,k+1}))$ with $1 \leq k < m$ are also elements of $CC_T^-(E)$. For this reason $(\, f(w_{1,1}, \ldots, w_{n,1}),\ \ldots,\ f(w_{1,m}, \ldots, w_{n,m})\,)$ is a valid transitive chain over $CC_T^-(E)$. □

Theorem 4. $D_m = CC_T(E)$. *[BN98, Theorem 4.3.5]*

Proof. By definition we already know $D_i \subseteq C_i \cap T \times T$ and as a conclusion $D_m \subseteq CC_T(E)$.
For the second case ($\supseteq$) let (w_1, w_n) be an arbitrary pair that is transitive in $CC_T^-(E)$ and $(w_1, w_2, \ldots, w_n)$ be the transitive chain for it. From Theorem 3 it follows that $(w_1, w_n) \in CC_T(E)$. The proof is now done by an induction over the pair $(n, \mathcal{N}(w_1))$, where n is the length of the transitive chain and $\mathcal{N}(w_1)$ is the nesting depth of w_1.

Induction base ($n = 1$): If $w_1 = w_n$, then $(w_1, w_n) \in D_1$ because of the reflexivity property of D and hence $(w_1, w_n) \in D_m$.

For the *induction step* we distinguish between two cases:

Case 1: For this case we assume that the chain contains a pair (w_i, w_{i+1}) $(1 \leq i < n)$ so that $(w_i, w_{i+1}) \in E$ or $(w_{i+1}, w_i) \in E$. Because the two sub-chains (1 to i and $i+1$ to n) are shorter than the whole chain (1 to n) and $w_i, w_{i+1} \in T$ the induction hypothesis implies $(w_1, w_i) \in D_m$ and $(w_{i+1}, w_n) \in D_m$. The statement $w_i, w_{i+1} \in T$ holds because all terms and subterms in E are also in T.
From (w_i, w_{i+1}) or $(w_{i+1}, w_i) \in E$ it follows that $(w_i, w_{i+1}) \in D_1 \subseteq D_m$ because $D_0 = E$ and D_1 additionally contains all symmetrical pairs of D_0. With (w_1, w_i), (w_i, w_{i+1}) and $(w_{i+1}, w_n) \in D_m$ the transitivity property can be used to conclude $(w_1, w_n) \in D_m$.

Case 2: In contrast to *Case 1* we assume the chain to be free of symmetrical pairs in E. In other words, no pair (w_i, w_{i+1}) $(1 \leq i < n)$ exists so that $(w_i, w_{i+1}) \in E$ or $(w_{i+1}, w_i) \in E$. This implies that every pair in the chain is made by congruence and for that reason they have some important properties:

1. w_i and w_{i+1} with $1 \leq i < n$ must have at least the same root function symbol: $w_i = f(t_{i,1}, \ldots, t_{i,o})$ and $w_{i+1} = f(t_{i+1,1}, \ldots, t_{i+1,o})$.

2. Because $w_1, w_n \in T$ and T is closed under subterms $t_{1,j}$ and $t_{n,j}$ with $1 \leq j \leq o$ are also in T.

3. $(t_{i,j}, t_{i+1,j}) \in CC_T^-(E)$ for all $1 \leq j \leq o,\ 1 \leq i < n$ and therefore $(t_{1,j}, \ldots, t_{n,j})$ $(1 \leq j \leq o)$ are valid transitive chains over $CC_T^-(E)$. Note that these chains of subterms have also a length of n.

4. For all $1 \leq j \leq o,\ 1 \leq i \leq n$ we are able to state about the nesting depth that $\mathcal{N}(t_{i,j}) < \mathcal{N}(w_i)$; especially for $i = 1$.

The properties 2, 3 and 4 allow to conclude by the induction hypothesis that $(t_{1,j}, t_{n,j}) \in D_m$ $(1 \leq j \leq o)$ and if this is the case, (w_1, w_n) must also be in D_m because of its congruence property. □

From Theorem 4 follows:

Corollary 1. *The word problem of ground identities is decidable.*

The *equality creation problem* can be reduced to the *word problem of ground identities*. This is done by using all equalities in the antecedent as the set of term pairs E. For the given term pair (s, t) we use the left and right side of the equality $s \doteq t$.

Corollary 2. *The equality creation problem is decidable.*

However, the result '*it is possible*' or '*it is not possible*' is not enough to solve a sequent. We need an algorithm instead that returns an exact list of instructions that allows to solve a given sequent. As a first step to achieve this, we must replace the concepts of symmetry, transitivity and congruence, which are used in the definition of *Cong*, by substitution because the Sequent Calculus, as defined in this thesis, has only rules for the concept of reflexivity and substitution to handle equalities. Consider the following redefinition of *Cong*:

$$\begin{aligned} Re(\hat{E}) &\coloneqq \{(u,u) \mid u \in G_\Sigma\} \\ Sub(\hat{E}) &\coloneqq \{(u,v) \mid \exists u', v', t_1, t_2.\ (u', v') \in \hat{E} \\ &\qquad \wedge ((t_1, t_2) \in \hat{E} \vee (t_2, t_1) \in \hat{E}) \\ &\qquad \wedge \exists p.\ (u'[t_1/t_2]_p = u \wedge v = v') \ \vee \ (v'[t_1/t_2]_p = v \wedge u = u')\} \\ Cong'(\hat{E}) &\coloneqq \hat{E} \cup R(\hat{E}) \cup Sub(\hat{E}) \end{aligned}$$

The definitions $Sy(\hat{E})$, $Tr(\hat{E})$ and $Co(\hat{E})$ are removed from *Cong* and $Sub(\hat{E})$ is added. The result of $Sub(\hat{E})$ contains all term pairs that can be created by substitutions over $\hat{E}$. A substitution is a replacement of two equal subterms. For such a substitution you need two term pairs: the first one (u', v') that is modified by a subterm replacement and a second one (t_1, t_2) to show that the original subterm and the new one are equal. With $Cong'$ we now define the sequence D' in the same way as it is already done for the sequence D but where $Cong'$ is used instead of *Cong*.

$$\begin{aligned} D'_0 &\coloneqq E \\ D'_{i+1} &\coloneqq Cong'(D'_i) \cap T \times T \end{aligned}$$

$D'_{m'}$ is again used to refer to the stable point of the sequence.

Theorem 5. $D_m \subseteq D'_{m'}$ [I]

Proof. We will show the correctness of this theorem by an induction over the sequence D. For the *induction base* we start with $D_0 = E = D'_0$. As the induction hypothesis we assume for a point D_i in the sequence D to have a corresponding point D'_j in the sequence D' such that $D_i \subseteq D'_j$. The *induction step* shows that each new element in D_{i+1} is also contained in some point D'_{j+x} $(x \in \mathbb{N})$ of the sequence D'.

[I] It may even be the case that $D_m = D'_{m'}$ but for the needs of this thesis it is sufficient to show that $D'_{m'}$ covers all the elements of D_m.

Symmetry: If $(v,u) \in D_i \subseteq D'_j$ and $(u,v) \in D_{i+1}$, then $(u,v) \in D'_{j+2}$ because (u,v) can be created by using two substitutions $(v[u]_\varepsilon, u[v]_\varepsilon)$.

Transitivity: If $(u,v),(v,w) \in D_i \subseteq D'_j$ and $(u,w) \in D_{i+1}$, then $(u,w) \in D'_{j+1}$ because (u,w) can be created by using a single substitution $(u, v[w]_\varepsilon)$.

Congruence: Let $\overline{x_n}$ be a short notation for a list of variables that covers n variables: $(x_1,\ldots,x_n)$. If $(u_1,v_1),\ \ldots,\ (u_n,v_n) \in D_i \subseteq D'_j$, $f^{(n)} \in \Sigma$ and $(f(\overline{u_n}), f(\overline{v_n})) \in D_{i+1}$, then $(f(\overline{u_n}), f(\overline{v_n})) \in D'_{j+n}$ because the reflexive pair $(f(\overline{u_n}), f(\overline{u_n}))$ can be transformed into the target pair within n substitution steps:

$$(f(\overline{u_n}), f(u_1[v_1]_\varepsilon, \ldots, u_n[v_n]_\varepsilon))$$

In simple words, symmetry, transitivity and congruence can be simulated by substitution and thus be replaced. □

Moreover, it is even possible to drop the reflexivity definition of $Cong'$ as shown in Theorem 2 if some requirements are met. Let $Cong''(\hat{E}) \coloneqq \hat{E} \cup Sub(\hat{E})$ be the congruence step where the reflexivity property is omitted and D'' the corresponding sequence. Additionally, let T' be the least set of terms that contains $\{u \mid (u,v) \in E \vee (v,u) \in E\}$ and is closed under subterms.

$$D''_0 \coloneqq E$$
$$D''_{i+1} \coloneqq Cong''(D''_i) \cap T' \times T'$$

Corollary 3. *If $s,t \in T'$ and $s \neq t$ then the sequence D'' can be used to decide the word problem of ground identities.*

The proof follows the one of Theorem 2. As long as all needed contexts are available for substitutions, which is the case if the terms s and t are already contained in T', all substitution steps can be done in place and there is no need of additional reflexive pairs. It is important to exclude the case where $s = t$ because every term is equal to itself independent of the given set of equal term pairs E. E.g. if $s \coloneqq c$, $t \coloneqq c$ and $E = \{(f(c), f(d)\}$, then (s,t) is not part of any D''_i $(i \geq 0)$ although $(s,t) \in T' \times T'$.
To handle cases where $s,t \notin T'$ or $s = t$ the decision can be done by using the following algorithm. Let $CC_{T'}(E)$ be the congruence closure that is calculated by the sequence D''.

As a result of Theorem 5 and Corollary 3 it is now possible to extend a sequent by using the contraction and substitution rules so that the set of equalities in the antecedent of the resulting sequent is closed under $Cong''$. This is important because it allows to decide the equality creation problem without the reduction to the word problem of ground identities.
However, the main goal is to decide the validity of an operator-free sequent with equality. This decision can be made by doing the following steps, while the Algorithm 3 is used to check if two terms can be made equal by substitutions:

Algorithm 3 Algorithm to decide the equality creation problem with $CC_{T'}(E)$

```
1: procedure CONTAINS((s,t) : G × G, CC_{T'}(E) : T' × T'): Boolean
2:     if (s,t) ∈ CC_{T'}(E) ∨ s = t then
3:         return true
4:     else if f^(n) ∈ Σ ∧ s = f(u_1,...,u_n) ∧ t = f(v_1,...,v_n) then
5:         return CONTAINS((u_1,v_1), CC_{T'}(E)) ∧ ··· ∧ CONTAINS((u_n,v_n), CC_{T'}(E))
6:     else
7:         return false
8:     end if
9: end procedure
```

Let $S \coloneqq (\Gamma, \Delta)$ be the given sequent, $E_c \subseteq \Gamma$ be the set of equalities in the antecedent that is already a congruence closure and $E_\Delta \subseteq \Delta$ the set of equalities in the succedent.

1. If a pair of formulas $(\varphi, \psi) \in \Gamma \times \Delta$ exists so that both formulas can be made syntactically equal by using substitutions on ψ, the sequent is valid.

2. If an equality $e \coloneqq (t_1, t_2) \in E_\Delta$ exists so that t_1 can be made syntactically equal to t_2 by using substitutions on t_2, the sequent is valid

3. If none of the two steps is applicable, the sequent is invalid.

4.2 Solving Quantifiers

Generally, quantifiers are solved by instantiation. Bound variables from left sided existential quantifiers and right sided universal quantifiers are replaced by a new constant. As there is nothing to choose or to decide this is the easy part of solving quantifiers. Left sided universal quantifiers and right sided existential quantifiers are different. You have to choose one or more ground terms for the instantiation in order to prove the sequent. If multiple instantiations of ground terms are needed per formula, the quantified formula has to be duplicated using the contraction rule first. Finding the right number and type of ground terms is, as the crucial part of First-Order Logic, a semi-decidable problem. The problem is already targeted in the study of *automated theorem proving* and it turned out it is already hard to semi-decide the validity problem for first-order formulas. As mentioned in Chapter 1 we have decided to solve quantifiers by using an existing SMT solver rather than implementing all from scratch. To do so we need to reduce the validity problem of sequents to a satisfiability problem of formulas. Multiple checks of the validity of modified sequents are then used to find applicable instantiations for quantified formulas. We first describe a basic and naive procedure of finding instantiations and then outline remaining problems which must additionally be considered. As the second step we show how to solve these problems and offer an algorithm that finds correct instantiations for mostly all sequents depending on the result created by

the SMT solver.

A sequent $S := (\Gamma, \Delta)$ is valid if all possible interpretations that satisfy all formulas from the antecedence also satisfy at least one formula in the succedent. And in contrast, a sequent is invalid if at least one interpretation satisfies all formulas out of the antecedent but none in the succedent. Theorem 1 is using this semantics to reduce the validity of sequents to the validity of formulas. Therefore, it allows to check the validity of sequents by using an SMT solver. The problem is that we generally cannot deduce some ground terms from the result of an SMT solver which can be used for a proof in the Sequent Calculus. However, when we have done some ground term instantiation first it is possible to validate if this instantiation was right or wrong. Given that, we are able to guess the right subset of ground terms using an enumeration over all ground term combinations. Unfortunately, already a small number of different function symbols may leads to a infinitely large set of ground terms and combinations of ground terms where complex ground terms can never be reached – in life time – by such an enumeration algorithm. For that reason we need an optimization of finding combinations of ground terms. This can be achieved by using the reduction to the validity problem as a heuristic which will dramatically reduces the search space.

Definition 16. *A partial instantiation is an instantiation where a context is used in place of a normal term. If Q is a quantifier, $Qx\ \varphi$ is a quantified formula and $f^{(n)} \in \Sigma$ is an n-ary function symbol, then $Qx_1, \ldots, x_n\ \varphi[f(x_1, \ldots, x_n)/x]$ is a partial instantiation of $Qx\ \varphi$.*

Partial instantiations can be used to make assumptions over the structure of a term in combination with SMT solvers. If a sequent is still valid after the partial instantiation with the function symbol f, we can conclude that f is the root symbol of the term that is needed for the instantiation. Because the number of function symbols is finite it is possible to iterate over all symbols and find the one that matches. So the idea of the Partial Instantiation Algorithm is to repeat the step of doing partial instantiations recursively for all new quantified variables $(x_1, \ldots, x_n)$ until the whole term is determined: as long as bound variables are left pick the next bound variable and try all available functions symbols as a candidate for a partial instantiation and keep the first valid sequent that is created by such a partial instantiation.
This first and naive definition for an algorithm still has some problems left. It does not define an ordering for the set of function symbols and it does not consider the case where multiple instantiations are needed. Certainly, we have to spend some extra effort to handle the case where multiple ground term instantiations are needed for the same formula and to do a proper enumeration over all instances of a formula and all function symbols accordingly. So the rest of this section covers exactly that. In the first place we describe the problem of multiple needed instantiations of ground terms, show some examples and explain how to solve the problem, whereas the second part is about an optimized enumeration over all candidates for a partial instantiation.

When multiple ground term instantiations are needed the sequent will be invalid for all

possible partial instantiations. See, for example, the sequents below. If the algorithm tries to solve the universal quantified formula before the conjunction on the right side is targeted, the algorithm has to consider two instantiations at the same time. In Section 4.3 we will see that the complete algorithm to prove sequents does exactly that. It prefers rule applications with a single premise over ones with two premises.

$$\forall x\ P(x) \implies P(f(f(c))) \wedge P(f(h(d))) \tag{1}$$
$$\forall x_1\ P(f(x_1)) \implies P(f(f(c))) \wedge P(f(h(d))) \tag{2}$$
$$\forall x_1\ P(f(f(x_1))) \implies P(f(f(c))) \wedge P(f(h(d))) \tag{3}$$
$$\forall x_1\ P(f(h(x_1))) \implies P(f(f(c))) \wedge P(f(h(d))) \tag{4}$$
$$P(f(c)) \implies P(f(f(c))) \wedge P(f(h(d))) \tag{5}$$
$$P(f(d)) \implies P(f(f(c))) \wedge P(f(h(d))) \tag{6}$$

The first and the second sequent are valid while the rest is not. In sequent 3 for example, it is not possible to conclude $P(f(h(d)))$ to be true from the premise $\forall x_1\ P(f(f(x_1)))$. So every time when all possible partial instantiations lead into invalid sequents, while the original sequent is valid, we may fork the current state (sequent 2) of the partial instantiated formula, because we expect to need multiple different instantiations to complete the proof.

$$\forall x_1\ P(f(x_1)), \forall x_1\ P(f(x_1)) \implies P(f(f(c))) \wedge P(f(h(d))) \tag{7}$$

With two instances it is now possible to partially instantiate one instance with f and the other one with h. Unfortunately, the partial instantiation will not work anymore for two or more instances of the same quantified formula if we just do it for one instance at a time. If only one of these formulas is modified, the sequent stays valid even for wrong partial instantiations. The next example demonstrates this behavior:

$$P(f(e)), \forall x_1\ P(f(x_1)) \implies P(f(f(c))) \wedge P(f(h(d))) \tag{8}$$

The first instance of the quantified formula is partially instantiated with a 0-ary function symbol – a constant –, which is indeed the same as a normal instantiation because no more quantified variables are left. This instantiation is wrong in the sense that we do not need and cannot even use the formula $P(f(e))$ to prove the sequent, even if the sequent is still valid. What we have done this way is – compared to sequent 2 – just adding a new formula to the sequent. Nothing else has been removed from the sequent and it is not possible to make a sequent invalid by just adding new formulas. As a consequence we need to do partial instantiations for multiple instances of the same formula at once if we want to obtain additional information about the searched term from that partial instantiation.

The Partial Instantiation Algorithm only terminates if constant function symbols are available and tried as instantiation candidates before all other function symbols are tried. The sequent 9, for example, allows to do infinite partial instantiations for the first quantified formula using the 1-ary function symbol f without making the sequent invalid. So if the naive algorithm mentioned above always tries f as the first candidate

for the next partial instantiation in this example, it ends up in a infinite loop.

$$\forall x\ P(f(x)) \implies \exists x\ P(f(x)) \tag{9}$$
$$\forall x_1\ P(f(f(x_1))) \implies \exists x\ P(f(x)) \tag{10}$$
$$\forall x_1\ P(f(f(f(x_1)))) \implies \exists x\ P(f(x)) \tag{11}$$
$$\forall x_1\ P(f(f(f(f(x_1))))) \implies \exists x\ P(f(x)) \tag{12}$$
$$\vdots$$

If the sequent does not contain a constant function symbol – like example 9 – it is possible to introduce a new constant because we can always define the signature τ for a given sequent so that it contains additional unused symbols. The order of function symbols that is used to do partial instantiation should then prefer constant function symbols as a candidate to ensure a proper termination.

An enumeration of candidates is an ordered list of all candidates for partial instantiations. Due to the fact that multiple partial instantiations are perhaps needed to do in parallel, candidates are tuples of function symbols with a varying length. Let F be the set of all available function symbols (G_Σ) increasingly ordered according to their arity. With reference to F every symbol can be mapped to a number $d \in \mathbb{N}$ by their position in F, where 0 is the first element in F and $k \in \mathbb{N}$ is the last element. With this mapping, a candidate $c \in C^n$ is an n-tuple of natural numbers, where C^n is the set containing all tuples with the length n. Doing a partial instantiation for a quantified formula φ with a candidate $c \in C^n$ means to create n instances of φ and do one partial instantiation for each instance using the function symbols that are defined by the candidate.

Definition 17. *C^* is the set of all candidates and is defined as*

$$C^* \coloneqq C^1 \cup C^2 \cup C^3 \cup \dots$$

A lot of candidates in C^* are equivalent in the context of partial instantiations. E.g. if two candidates contain the exact same set of numbers but with a different order, the set of formulas that results from doing partial instantiation for the two candidates are the same. Another example, if a candidate $c_1 \in C^n$ contains a number $d \in \mathbb{N}$ twice, it is equivalent to the candidate $c_2 \in C^{n-1}$ that contains the same elements as c_1 but with only one instance of d. For this reason we define a new set of candidates that is free of equivalent pairs.

Definition 18. *A candidate $a \coloneqq (a_1, \dots, a_n)$ is equivalent to a candidate $b \coloneqq (b_1, \dots, b_m)$ – written as $a \sim b$ – iff for each element a_i $(1 \leq i \leq n)$ there is an element b_j $(1 \leq j \leq m)$ with $a_i = b_j$ and vice versa.*

Definition 19. *$C^*_\sim$ is set that contains exactly one instance per equivalence class:*

$$C^*_\sim \coloneqq \{(c_1, \dots, c_n) \in C^* \mid \forall i, j.\ i < j \to c_i > c_j\}$$

So every tuple in $C^*_\sim$ is made of pairwise distinct elements which are decreasingly ordered.

Definition 20. *A candidate* $a \coloneqq (a_1, \ldots, a_n)$ *is less than a candidate* $b \coloneqq (b_1, \ldots, b_m)$ *– written as* $a <_c b$ *– iff* $n < m$ *or if* $\exists i.\ a_i < b_i \wedge \forall j\ j < i \rightarrow a_j = b_j$.

In other words $<_c$ is a lexicographic ordering of candidates. For a given set of candidates C the $<_c$-successor of an element a is the least element that is greater than a and is contained in C: $next(a, C)$. With this successor definition and the set of distinct candidates $C^*_\sim$ we are now able to define the final enumeration as the sequence E:

$$E_0 \coloneqq (0)$$
$$E_{i+1} \coloneqq next(E_i, C^*_\sim)$$

Algorithm 4 Enumerating of function symbols

```
1:  procedure NEXT(candidate: List, highestSymbol: ℕ): Boolean
2:      var length ≔ LENGTH(candidate)
3:      for i ≔ length − 1 down to 0 do
4:          var rowMax ≔ highestSymbol − (length − i − 1)
5:          if (candidate[i] ≥ rowMax) then
6:              if (i == 0) then
7:                  return false;
8:              else
9:                  continue                    ▷ Jump to the next iteration
10:             end if
11:         end if
12:         candidate[i] + +
13:         var nextValue = candidate[i] + 1
14:         for j = i + 1 to j < length  do
15:             candidate[j] = nextValue
16:             nextValue++
17:         end for
18:         break;                                ▷ Leave the loop
19:     end for
20:     return true;
21: end procedure
```

Legend

:	is used to define types
≔	is used as the assignment operator
[ ]	is used as the list operator.
++	is used as the increment operator

An example implementation of the successor function is given by Algorithm 4. The algorithm accepts a tuple in form of a list and calculates the next successor. In the sense of mutable data structures the algorithm modifies the given list instead of returning a new one. The return value however is used to indicate the need of a larger list. This allows to handle the duplication part outside the successor algorithm.

The idea of doing partial instantiations recursively for all bound variables in combination with the enumeration given by the sequence E can be used to formulate an algorithm that searches for a proper instantiation for one specific quantified formula within one specific sequent. But as outlined in Section 2.2, it is only semi-decidable if a first-order formula is unsatisfiable. This is why the SMT solver, which is used to decide if a sequent is valid (see Theorem 1), may run infinitely long for some sequents. To handle such cases we must introduce a timeout for the SMT solver. If the SMT solver exceeds the timeout, the calculation is stopped and the result is treated as *unknown* – thus possible return values are *valid*, *invalid* or *unknown*. Obviously, it is not possible to conclude that a sequent is invalid if the SMT solver hits the timeout because the solver may find a solution with a larger timeout. These unknown-results must additionally be handled in some way by the algorithm. Unfortunately, the algorithm itself may also run infinitely long. This is the case when the SMT solver has returned unknown for a critical set of sequents or if the given main sequent was invalid. To ensure the termination of the algorithm we define a maximum number of contractions that are allowed for one formula (called the contraction limit). If the algorithm reaches the contraction limit it should stop and return itself the result *unknown*. Figure 4 shows the Partial Instantiation Algorithm[I] that covers all that: it uses the sequence E to find instantiations for bound variables, it recursively eliminates all bound variables, it handles cases where the underlying SMT solvers returns *unknown* and it allows to set a contraction limit. Unknown-results of the SMT solver are handled as follows:

- If the validity-state of the incoming sequent is *unknown* the algorithm assumes the sequent to be valid. This is no problem because a wrong assumption leads to the same result: no term found.

- If the SMT solver returns *unknown* after a partial instantiation has been done, the algorithm assumes the sequent to be invalid. But if this assumption was wrong, the algorithm may not be able to find the right instantiation anymore. For this reason the algorithm switches over to a special set of states to consider the unknown-result received before. As soon as the sequent becomes valid again, it is clear that the assumption was correct and the algorithm returns back to normal.

[I] The meaning of the state "change variable focus" that is shown in Figure 4 will be explained in detail in Chapter 5.

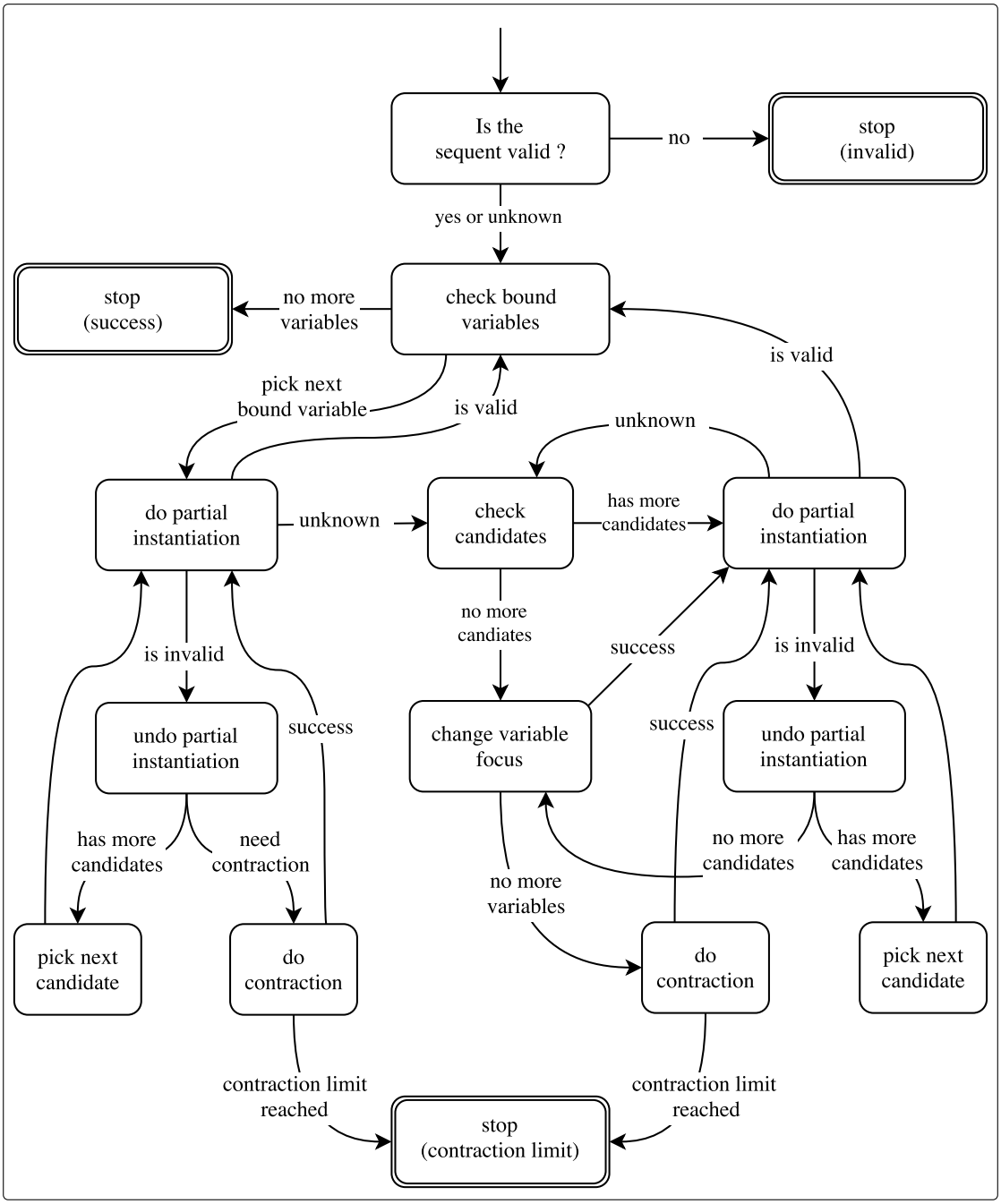

Figure 4: A block diagram showing all the steps that are related to the Partial Instantiation Algorithm.

4.3 The Term Guessing Algorithm

Algorithm 1 describes how to decide the validity problem for sequents without quantifiers and without equalities. Subsection 4.1 and Section 4.2 introduced additional algorithms to handle equalities and quantifiers. The last step is to combine all the results to create an algorithm for sequents over the complete set of first-order operators. Because of the special handling of quantifiers we call it the Term Guessing Algorithm.
Algorithm 1 tries to apply rules in an unspecified order. This is suitable if the resulting proof does not matter, but the aim of this thesis is to create short proofs and the order of rule applications is one of the main influencing factors for the size of a proof. Unfortunately, there is no static order of rules that always leads to the smallest proof compared to all other rule orders when the size of a proof is measured by the number of sequents. For the following two sequents it is even impossible to state one preference order for the rules $\wedge_L$ and $\vee_R$ that leads for both sequents to the smallest proof:

$$P \wedge Q \implies Q \vee P, P \tag{1}$$

$$P \wedge Q, P \implies Q \vee P \tag{2}$$

For sequent 1 it is necessary to use the $\wedge_L$-rule first, which makes it possible to apply the axiom rule afterwards. In contrast, if the $\vee_R$-rule is preferred and solved first, the resulting proof would have one additional sequent. However, for the second sequent the order must be the other way round to achieve the smallest proof. An algorithm that analyses the structure of a sequent may be able to calculate the best application order of rules for a given sequent, but for complex sequents this may take a long computation time. With the aim of creating human-readable proof it would be sufficient to produce just small – and not the smallest – proofs in all cases. This applies to the simple order that prefers rules with a single premise over the ones with two premises and that searches for applicable axioms in every single step.
Figure 5 shows the main loop of the Term Guessing Algorithm. Within each iteration of the main loop the sequent tree is checked for unproved sequents, while the first unproved sequent is picked. As an alternative, unproved sequents can be organized in a FIFO-queue. The main loop is made up of 5 phases and in each phase the algorithm tries to solve a specific set of operators.

- In the Axiom Phase the algorithm looks for applicable axiom rules.
- The single premise phase covers the rules $\neg_L, \neg_R, \wedge_L, \vee_R, \rightarrow_R, \exists_L$ and $\forall_R$.
- The Term Guessing Phase uses the result of Section 4.2 to find instantiations for quantified formulas. If it fails for one formula it tries another one until it succeeds or no more formulas are left. Because it is possible that the Partial Instantiation Algorithm runs infinitely for a given formula we use a limit on the number of contractions the algorithm is allowed to do for one formula. This ensures a proper termination after a finite number of steps.
  Note that the rules $\exists_L$ and $\forall_R$ are already handled in the single premise phase.

- The Double Premise Phase does the same as the single premise phase but for all the rules with two premises: $\vee_L, \wedge_R, \rightarrow_L, \leftrightarrow_L$ and $\leftrightarrow_R$.

- The last phase is the Substitution Phase. As outlined in Section 4.1, a proper equality handling is not possible until all other operators are solved. For this reason the substitution phase is only entered if all other phases have failed.

Every phase is intended to resolve only one formula per run. If this is done, the algorithm goes back to the beginning and searches for applicable axiom rules. A phase fails if it is not possible to resolve any formula that corresponds to the associated set of rules. If all phases have failed in a single run, the algorithm stops with the result *unknown* or *invalid*. The result depends on the sequent. If any unproved leaf-sequent of the resulting proof tree still contains at least one quantified formula, it is not possible to conclude that the sequent is invalid. But if all quantifiers are solved at this state or if the original sequent was quantifier free, the sequent must be invalid. This is a conclusion of the decidability of sequents over $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \doteq\}$.

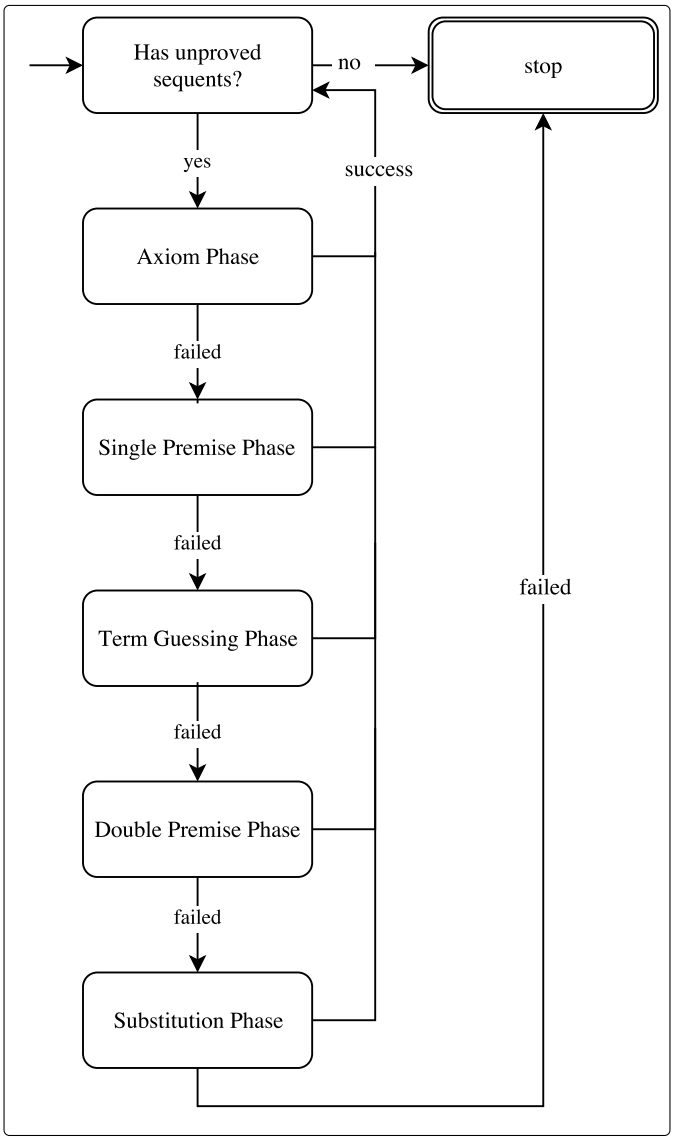

Figure 5: A diagram showing the main loop of the Term Guessing Algorithm.

5 Implementing the Term Guessing Algorithm

In contrast to Chapter 4, which elucidates the theoretical basis for the Term Guessing Algorithm and the Term Guessing Algorithm itself, this chapter is focused on some implementation-specific decisions and on further improvements – especially on the number of used substitutions.
To summarize the results of Chapter 4, the following list of algorithms and procedures may be used for an implementation of the Term Guessing Algorithm:

- The main loop of the Term Guessing Algorithm may follow the one shown in Figure 5.
- Quantifiers can be solved by using the Partial Instantiation Algorithm shown in Figure 4, while Algorithm 4 should be used to enumerate instantiation candidates.
- Equalities can be handled by following the steps on page 30. To do so, the Algorithm 3 is used to decide the equality creation problem and the sequence D'' on page 32 is used to calculate the congruence closure $CC_{T'}(E)$. As an alternative to D'' the Bucket Combination Algorithm (Algorithm 5) may be used, which will be introduced in this chapter.

Additionally, the Partial Instantiation Algorithm requires an SMT solver for its calculations. Some possible solvers are listed in Chapter 1. In practice, Microsoft Research's Z3 Prover stands out due to its special handling of some decidable fragments of FO[=], its bindings (API) for various programming languages and its ongoing further development. In the context of the programming language Java the Z3 Prover is currently one of the best choices.

The following two sections are further improvements to speed up the search for instantiations and to reduce the proof size for sequents with equality. An implementation that follows the steps mentioned above may be complemented by theses improvements.

5.1 Faster Instantiations

Section 4.2 introduces a timeout and a contraction limit for the Partial Instantiation Algorithm. The timeout is set as the maximum computation time for each run of the underlying SMT solver. A lower timeout may increase the speed of the algorithm because infinitely long runs of the SMT solver are canceled sooner, but it also increases the chance of exceeding the timeout for all other runs. Therefore the timeout should meet the properties of the formula. For large formulas a long timeout should be applied, while short formulas do not need much computation time in general. The best timeout depends on a lot of other factors like the used hardware and must be determined by test runs on a representative set of formulas. The same holds for the contraction limit.

The contraction limit is the maximum number of allowed contractions per formula. It ensures the termination of the Partial Instantiation Algorithm in some cases – e.g. if the algorithm receives too many unknown-results from the SMT solver and thus fails to identify the correct solution. Similarly to the timeout, a lower contraction limit improves the speed of the algorithm but if it is set too low, the right solution cannot be found. Imagine the case where a formula needs three different instantiations in order to prove the sequent but the contractions limit is set to two. This is why the number of needed contractions per formula should be estimated for the target set of formulas.

While using the Z3 Prover it turned out that changing, adding or removing only one function symbol can change the result from *valid* or *invalid* to *unknown* and the other way round. Based on this experience we added the step of skipping variables to the Partial Instantiation Algorithm. The step is already contained in the diagram shown in Figure 4: "change variable focus". When the Partial Instantiation Algorithm reaches the point where no more candidates with the current length are left it uses contraction and continues to search for applicable partial instantiations (see the left part of Figure 4). But when the algorithm already received unknown-results it is unclear if a contraction is really needed. This is where the variable skipping applies. If more than one bound variable is at hand, it is possible to skip the current variable and set the focus on another one. The previous bound variable may become solvable if one or more partial instantiations for other bound variables are successfully applied. Only when no more bound variables are left the algorithm falls back to the contraction strategy.

5.2 Fewer Substitutions

Congruence closures can become very large sets even in combination with the restriction T or T' (see Section 4.1). This is why a congruence closure would blow up any proof in the Sequent Calculus with a bunch of substitution applications. But not all equalities that are introduced by the congruence closure are needed to prove the sequent. Obviously all unused equalities can be removed, but we do not know which ones are unused until the congruence closure and the term comparison were made. The result of the term comparison is a couple of equalities that are used to complete the proof (the target equalities E_t). Starting from these equalities in E_t we can compute the set of all needed equalities out of the whole congruence closure by a backward search.
The final version of the sequence that is used to calculate the congruence closure (D'', as defined in Section 4.1) uses substitutions only. For these substitution steps we need two equalities that are already available: one equality that is modified by a term replacement and another one that legitimates the replacement. In order to refer to these equalities we introduce some metaphoric names. We call the modified equality the main ingredient, the legitimization equality is called the auxiliary ingredient and both together – in combination with the replacement instructions – is a recipe.
For the purpose of doing a backward search the congruence closure algorithm has to log

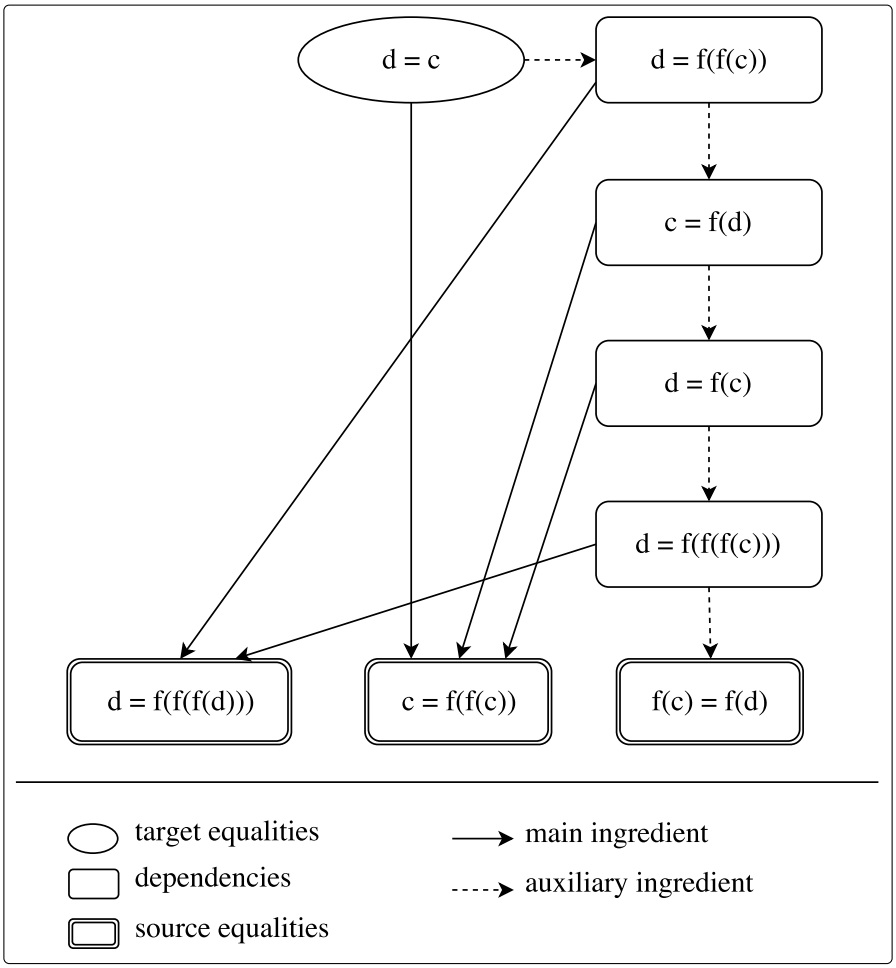

Figure 6: An example of a dependency graph for equalities. The target equality $d \doteq c$ is built using three source equalities.

the recipe for every single equality, which is easy to implement. With this log we create a dependency graph that starts with all target equalities and ends up with the source equalities ($E_s = E$) as leaf nodes. Figure 6 illustrates an example dependency graph where all dependencies of the equality $d \doteq c$ are listed down to the leaf nodes, which do not have dependencies. The graph has to be a minimal DAG[I], where each node is unique.

Instead of doing the full congruence closure within the sequent, we may do the following to reduce the size of the proof tree:

1. Extract all equalities E out of the antecedent and create a congruence closure from it by using $CC_{T'}(E)$ outside the sequent, while logging all the recipes.

[I] A DAG is a directed acyclic graph

2. Check which equalities E_t are needed (target equalities) to prove the sequent.

3. Do a backward search for each equality in E_t using the logged recipes. Follow the dependencies of each recipe down to the source equalities E.

4. Create a combined DAG for all equalities in E_t and all their dependencies.

5. Use substitution and contraction rules so that the resulting sequent contains all target equalities E_t using the DAG as a step-by-step instruction.

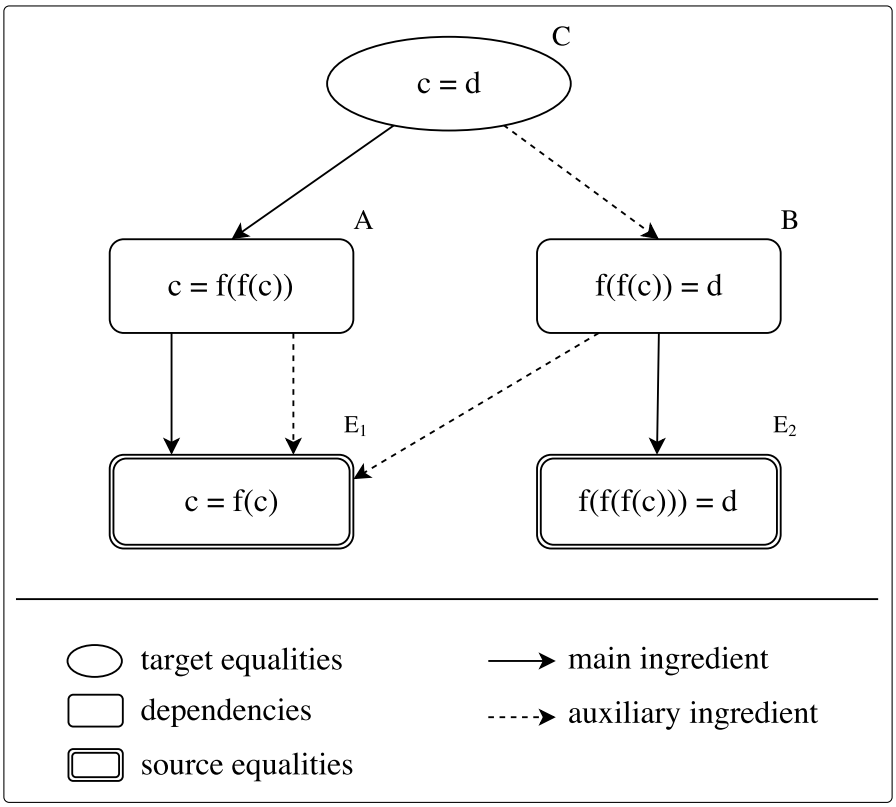

Figure 7: An example of a dependency graph for equalities. The target equality $d \doteq c$ is built by using two source equalities.

For the last point you have to consider the behavior of the substitution rules. In contrast to the definition of substitutions in D'', where term pairs are never dropped, the substitution rules of the Sequent Calculus modify (consume) the main ingredient. This is why main ingredients have to be duplicated by using the contraction rule, when they are used multiple times within the DAG. One approach is to use the contraction rule every time before a main ingredient gets modified, but this is not necessary in some cases and the resulting proof tree should be as small as possible. The usages of the contraction rule can be reduced by an execution order of recipes, where recipes are executed first if their main ingredient is not a target equality and does not have other references.

Figure 7 shows an example DAG of recipes where the execution order B, A, C allows to accomplish all recipes without duplication, while the order A, B, C requires the equality E_1 to be duplicated because otherwise recipe B is missing its main ingredient.
We can calculate the best execution order of recipes by a depth-first search with backtracking but such an algorithm has an exponential complexity. These costs are out of proportion with the goal to reduce the number of used contraction rules. Instead we use a heuristic to achieve a good – but not always the best – solution, which results in a linear complexity.

Let *Todo* be the set of recipes that are still unfinished and $\mathit{next}(\mathit{Todo})$ be the set of unfinished recipes where all dependencies are already available.

1. If $\mathit{next}(\mathit{Todo})$ is empty: stop.

2. Accomplish all recipes $r \in \mathit{next}(\mathit{Todo})$ where the following conditions are met: the main ingredient is not a target equality and has exactly one reference (the reference to r). On success go back to step 1.

3. Accomplish *one* recipe $r \in \mathit{next}(\mathit{Todo})$ where the following condition is met: the main ingredient is referenced more than once as a main ingredient or is a target equality. On success go back to step 1.

4. Accomplish the most referenced recipe $r \in \mathit{next}(\mathit{Todo})$. Repeat step 1.

The current definition of D'' describes exactly its output per step but an order for the substitutions ist still missing, which is very important for every resulting DAG. If the execution order of some substitution steps is unfavorable within the congruence closure, the resulting DAG for a given target equality can become very large. For further minimizations of the proof tree we want the dependency DAG for every equality in the congruence closure to contain as few nodes as possible. Algorithm 5 is an implementation of $CC_{T'}(E)$ that ensures the smallest step count for every new equality.
The step count for a equality e is the number of substitutions that were made to create e. If e is a source equality, then $\mathit{steps}(e) \coloneqq 0$. If m_e is the main ingredient of e and a_e is the auxiliary ingredient of e, then $\mathit{steps}(e) \coloneqq \mathit{steps}(m_e) + \mathit{steps}(a_e) + 1$.
What is still missing for the algorithm is the restriction T' that is applied to every step of $CC_{T'}(E)$. The restriction allows to drop every new equality that is not contained in $T' \times T'$, which requires the algorithm to compare every new equalities with all terms in T'. This becomes ineffective very quickly for large sets T'. To work around that performance issue we can make a compromise between computation time and memory usage.

Definition 21. $T_{\mathcal{N}} \coloneqq \{t \mid t \in G_\Sigma \wedge \mathcal{N}(t) \leq h_{max}\},\ \ h_{max} = \max\{\mathcal{N}(t) \mid t \in T'\}$.

$T_{\mathcal{N}}$ contains at least all elements in T' and can therefore be used as an alternative

restriction for the calculation of the congruence closure. To check if a term t is an element of $T_{\mathcal{N}}$ you just have to calculate $\mathcal{N}(t)$. This is much faster than a comparison with a whole set of terms. The disadvantage – compared to the restriction T' – is the size of $T_{\mathcal{N}}$, which is much higher than the one of T' and so is the resulting congruence closure. As mentioned before, this is a compromise with the result of a higher memory usage but a lower computation time.

On top of the step-count definition and the height restriction $\mathcal{N}(t)$ we introduce the following algorithm that minimizes the step count for each equality.

Algorithm 5 Bucket Combination Algorithm

```
1:  procedure BCA(E)
2:      var buckets := [{E}].                                  ▷ a list of sets
3:      var i := 0
4:      while i < LENGTH(buckets)) do
5:          Remove all elements e from buckets[i] where e or S(e) is already con-
            tained in buckets[j] with j < i.
6:
7:          for all e ∈ buckets[i] remove S(e) from buckets[i]
8:
9:          for j := 0 to i do
10:             var newEqualities := combine all in buckets[j] with all in buckets[i]
                                     and vice versa.
11:             newEqualities := newEqualities ∩ T_N × T_N
12:             buckets[j + i + 1] := buckets[j + i + 1] ∪ newEqualities
13:         end for
14:         i := i + 1
15:     end while
16:     return buckets
17: end procedure
```

Legend

$:=$	is used as the assignment operator
[ ]	is the list operator.
{ }	is the set operator.
$S(e)$	symmetrical equivalent version of e

Algorithm 5 classifies all equalities by their step count. It starts with bucket 0 that contains all equalities with a step count of 0 – the source equalities. Because it is impossible to create additional equalities with a step count of 0 we say *bucket 0 is closed*. The algorithm combines all equalities from closed buckets with all the other ones

from closed buckets. In the first iteration all of bucket 0 is combined with all of bucket 0, which may result in a couple of equalities for bucket 1 ($(0 + 0) + 1 = 1$). Because bucket 0 is closed, and all combinations that may create equalities with a step count of 1 are already proceeded, bucket 1 will be closed after the first iteration. In general, we can assume bucket $i + 1$ to be closed after the i-th iteration is done, where 0 is the first iteration. Similarly to iteration 0 iteration 1 combines all closed buckets with bucket 1 (including itself). The combination of bucket 1 with bucket 0 produces equalities with a step count of 2 ($(1+0)+1 = 2$), whereas the combination of bucket 1 with itself produces equalities with a step count of 3 ($(1+1)+1 = 3$). To ensure the lowest step count for all equalities the algorithm removes equalities from a bucket if they are already contained in another bucket with a lower step count. This is done for every bucket after it has been closed. The algorithm stops when all buckets are closed. Figure 8 demonstrates the first 3 iterations.

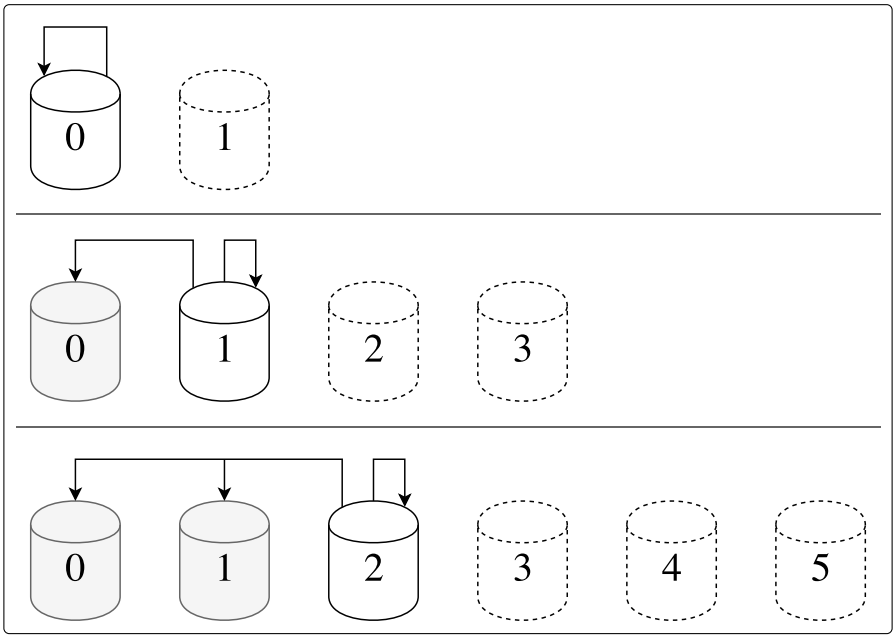

Figure 8: The image is showing the first three iterations of the Bucket Combination Algorithm. Dotted buckets are open ones that can still be filled by the algorithm. Solid buckets are closed.

6 The Sequent Calculus Trainer

The Sequent Calculus Trainer is a cross-platform desktop application and it is made to support students in learning the Sequent Calculus. It is a project by the department *Theoretische Informatik/Formale Methoden* at the University of Kassel and was funded by the *Service Center Lehre* as part of the educational initiative *Lehrinnovationen*[I]. [EHL15]

The first version from 2013, which is shown in [EHL15], was able

- to indicate syntactical mistakes,
- to give hints when the user tries to incorrectly apply rules and
- to offer short descriptions about how to correctly apply rules.

With the results of the previous sections the Sequent Calculus Trainer version 3.0.0 has become smart. It is now able to automatically find proofs. After it has found such a proof in the background it is able to give hints to the user.

The following screenshots demonstrate the old and new features of the Sequent Calculus Trainer:

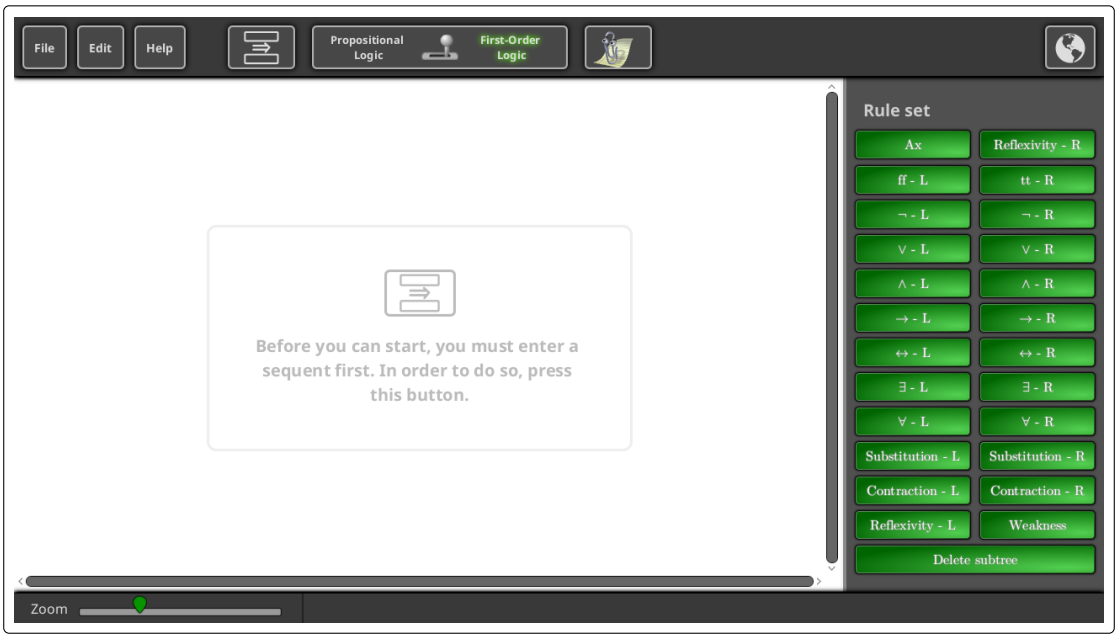

Figure 9: Screenshot of the Sequent Calculus Trainer version 3.0.0 showing the main window after the application start.

[I] Innovations in teaching

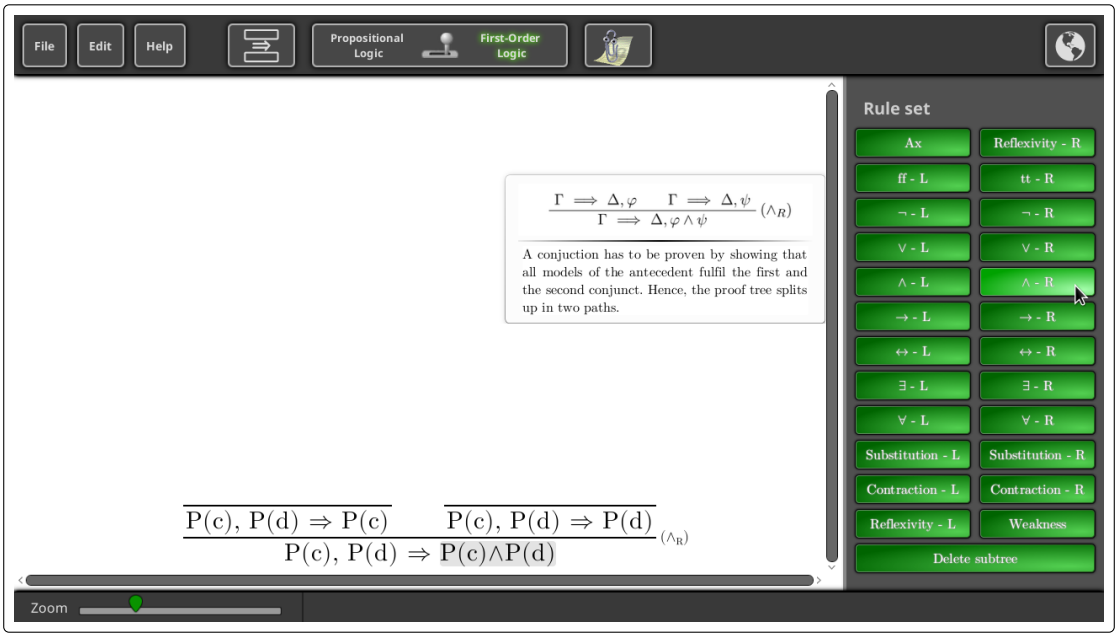

Figure 10: Screenshot SCT 3.0.0: showing a popover with further information about how the not-left rule works.

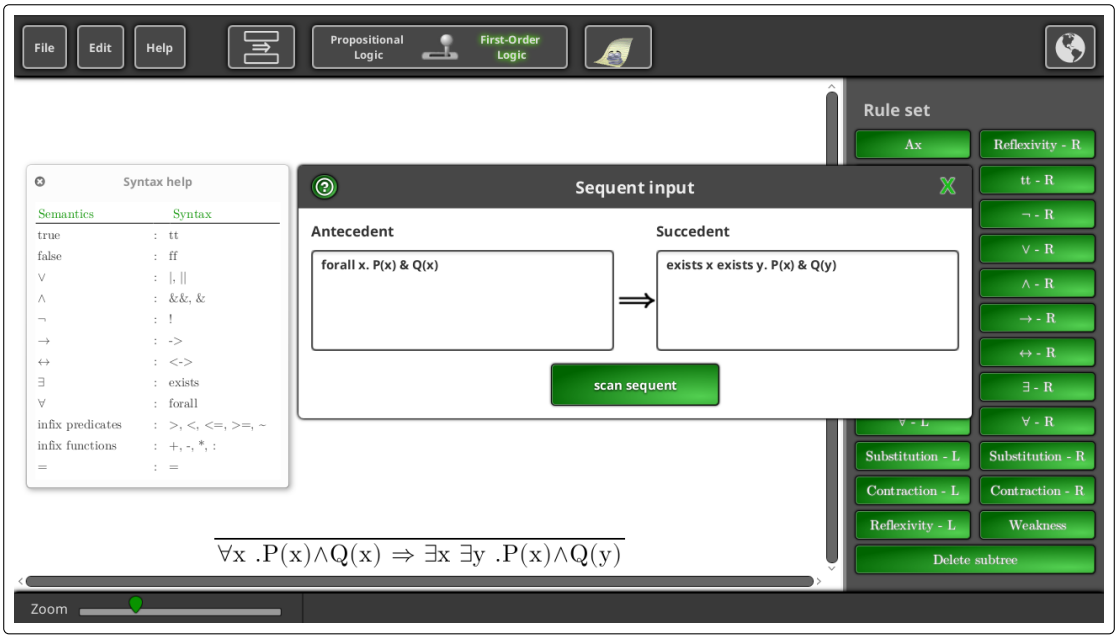

Figure 11: Screenshot SCT 3.0.0: showing how to enter a new sequent. An extra window with a documentation about the syntax opens by clicking the question mark button.

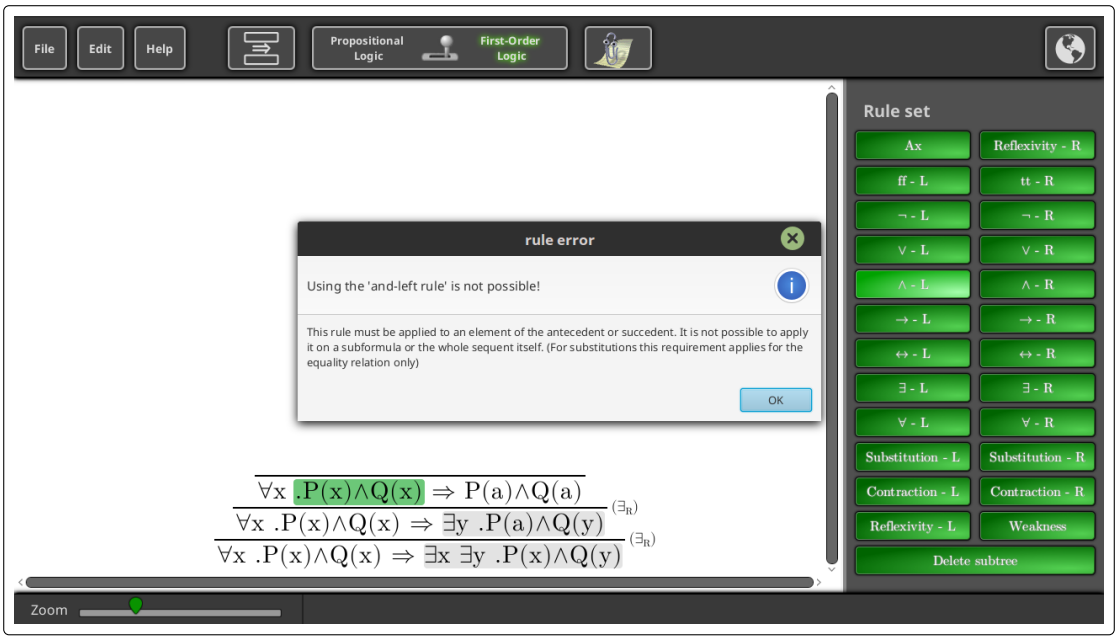

Figure 12: Screenshot SCT 3.0.0: showing an error popup that tells the user what he did wrong.

Figure 13: Screenshot SCT 3.0.0: showing a complete proof for one of the example sequents.

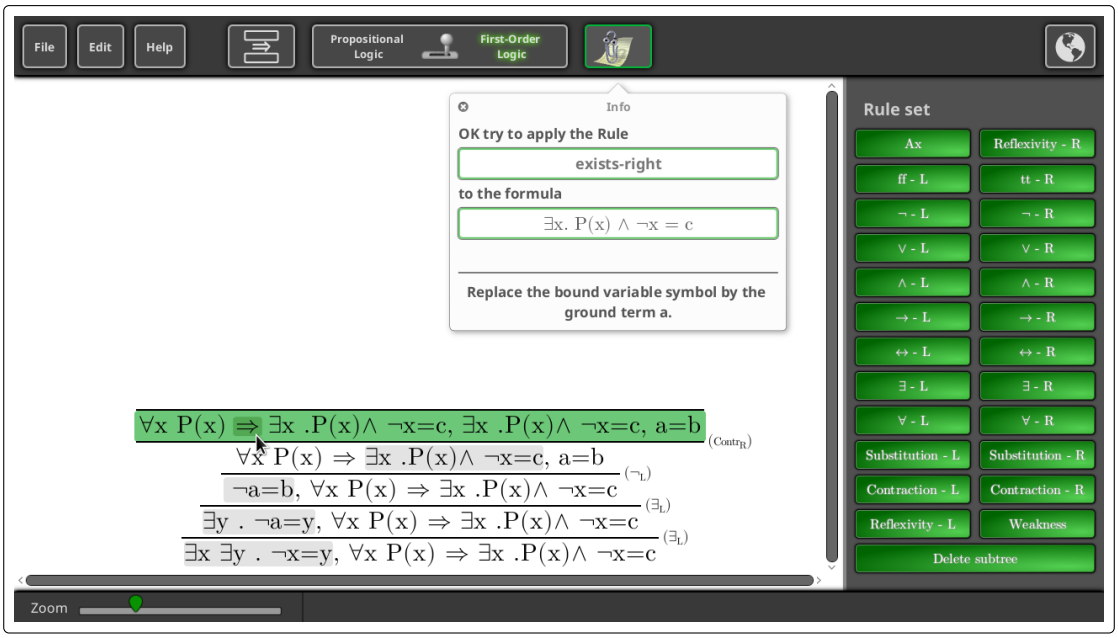

Figure 14: Screenshot SCT 3.0.0: showing the SCT while the support mode is turned on and the user asked for help with the last sequent.

Figure 15: Screenshot SCT 3.0.0: showing the SCT while the support mode is turned on. The SCT marked all unsolved sequents with a green or red background indicating that the sequent is still valid or if it has become invalid.

6.1 How to use it

The general workflow of the Sequent Calculus Trainer 3 is as follows. You open the *Sequent Input Scene* and enter a formula using the ASCII-syntax (see Figure 11). After pushing *scan sequent* your sequent will be displayed at the bottom of the screen.
The next step is to push a rule button of your choice and to click a formula to apply the chosen rule. If your rule application is syntactically and structurally correct, the current sequent gets one or two subsequents according to the chosen rule. You may continue to apply rules to create a complete sequent tree. The proof is done if an axiom has been applied to all leaf sequents of the created sequent tree.

At any time you may activate the support mode. This is done by clicking on Clippy. Clippy is the small paperclip in the top center. It has a surrounding green border if the support mode is active and an orange one if Clippy is busy. While the support mode is on, all sequents are marked with a red, yellow or green color indicating that the sequent is unsolvable (invalid), the algorithm is not able to make a decision (unknown) or the sequent is solvable (valid). If a sequent is marked green and its subsequent turns red after a rule application, the user may recognize that this rule application was wrong.
If you hover over Clippy while the support mode is turned on, Clippy will offer its help to you. Confirm and click a sequent where you need help. Clippy will tell you what step to do next. To get more help, repeat these steps.

Sometimes you just want to get a sequent proved. For this purpose push the *edit* button and choose *proof sequent*. When the calculation has finished the current sequent tree will be replaced by the calculated proof. But the current sequent tree is used as input for the proof search algorithm. If you have already created an invalid subsequent, the algorithm will fail even if the root sequent is provable by the algorithm. In this case just reset the sequent tree and trigger the proof search again.

Another useful feature is the image export function. Click *file* and choose *export proof as picture* and the current proof area of the Sequent Calculus Trainer will be converted to a *png*-image. If the proof is much smaller than the current proof area you may resize the window to fit the proof before you create the picture. This saves time with post-processing the image.

7 Conclusion

In this thesis we have shown that the Sequent Calculus over $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \doteq\}$ is decidable by a reduction of the *equality creation problem* to the *word problem of ground identities* which was solved by a finitely calculable congruence closure. Moreover, we have shown how to extract the shortest creation path for each equality out of a congruence closure and provided an algorithm to map all the steps to a suitable and small set of rule applications. In Section 4.2 we have shown how to solve quantifiers within sequents by using an SMT solver. For this purpose we used a reduction of the validity problem of sequents to the validity problem of formulas. The SMT solver was then used to verify assumptions over the root function symbol of a searched term. In Section 4.3 we have created the Term Guessing Algorithm by the combination of the algorithm for equalities, the algorithm for quantifiers and the general algorithm for sequents over propositional logic.
An example of the Term Guessing Algorithm has been implemented as a proof of concept in the Sequent Calculus Trainer, which is a program for students to learn the Sequent Calculus. With this algorithm, the Sequent Calculus Trainer is now able to automatically prove sequents in the background and provide tips and hints to users that struggle with finding proofs.

One of the main objectives was to create human-readable proofs. For this reason a lot of additional work has been done to reduce the number of subsequents. The greatest effect on the number of subsequents was achieved by the search for the smallest creation path of equalities and by the ordering of rule applications in the main part of the algorithm. For most of our test-sequents the algorithm finds one of the smallest possible proofs and in all other cases the generated proof contains just a few more rule applications compared to the minimum (see Table 1). This is a great result for the Sequent Calculus Trainer which will highly increase the understanding of generated proofs and thus its effectiveness in teaching the Sequent Calculus.

Another main object was to make the algorithm as fast as possible for small and medium-sized sequents. The computing speed for the Term Guessing Algorithm mostly depends on the underlying SMT solver. The Z3 Prover is one of the best SMT solvers and as such its computation time on small formulas is most of the time not even measurable. But for some and even small formulas the Z3 Prover is unable to find a solution and runs indefinitely. These cases were solved by a configurable timeout and a proper handling of unknown-results by the Term Guessing Algorithm. Because a lower timeout increases the computation speed it is a good idea to choose a very small timeout after the SMT solver is stopped. This timeout must be sufficient for the used input sequents. The default setup for the Sequent Calculus Trainer is currently 500ms for each run of the SMT solver.
Our experience is that the Z3 Prover fails for formulas that contain terms with a lot of different function symbols and a large nesting depth. The smallest example that we

found and for which the Z3 Prover (version 4.5.0) runs indefinitely is

$$\forall x\ P(f(g(f(x)))) \wedge \neg P(f(g(h(c))))$$

The Z3 Prover offers a set of options for resolving terms. By default, the Z3 Prover calculates a set of patterns for each quantifier that is used to find instantiations. A pattern defines a subset of applicable instantiation candidates. An empty set of candidates indicates an unsatisfiable formula. If the set of candidates is still infinite, the Z3 Prover tries the model-based instantiation method where terms are built and checked iteratively. The Z3 Prover can also handle a set of decidable fragments of First-Order Logic like the class of effectively propositional formulas. Thus, if a formula is not in one of the decidable fragments known by the Z3 Prover and uses a combination of terms that makes it hard to find a good pattern for, there is a high chance for the formula to be unsolvable by the Z3 Prover.

A second fact that influences the speed of the algorithm is the size of the calculated congruence closure. Because the congruence closure is limited by the nesting depth of terms the congruence closure grows with the number of the highest nesting depth. However, in most cases, not the time but the space will be the critical factor for the calculation of congruence closures. The final size of calculated congruence closures could be further improved by using the restriction T' instead of the restriction $T_{\mathcal{N}}$, which is currently implemented by the Sequent Calculus Trainer (see Section 4.1 for an explanation about T' and $T_{\mathcal{N}}$). This improvement is at the expense of a larger computation time, which becomes acceptable if you use a better term comparing algorithm for term sets. As outlined in Section 4.1, the restriction T' requires the algorithm to compare all new terms with all terms in T'. One idea to improve this comparison is to calculate a combined tree representation of all terms in T' in the beginning of the calculation. This combined tree representation can then be used to compare new terms with all terms in T' at once.

The results of some test-sequents for the Term Guessing Algorithm are shown in Table 1. The examples 1 to 6 are quantifier- and equality-free and thus are solved in no time – ~ 0 indicates that the time was below 0.1 seconds and therefore not measurable by our test system. The examples 7 to 11 are made with equalities and the calculated congruence closure grows from example to example which leads to a higher calculation time. Note that example 11 is basically the same as example 10 but with an additional useless formula that dramatically increases the highest nesting depth. This causes the congruence closure to grow to a multiple of the one from example 10. The examples 12 to 16 contain quantifiers, that is why the algorithm needs one or more calls of the underlying SMT solver to solve them. The very first call of the Z3 Prover needs some additional time for the Z3 Prover to be initialized, which takes about 0.1 seconds. The rest of the algorithm takes again no time for these examples. The examples 17 and 18 represent a combination of quantifiers and equalities. Example 18 seems to be simple at the first glance, but requires at least one contraction of the existential quantified formula on the right side to be solved. With a proof size of 18 the algorithm found the smallest possible proof for example 18. The last three examples are included to demonstrate the

Table 1: Calculation times for the Java-Implementation of the Term Guessing Algorithm in the Sequent Calculus Trainer

No.	Formula	Proof size(1)	Time(2)
1	$A \implies A$	1	~ 0
2	$A, B \implies A \wedge B$	3	~ 0
3	$(A \leftrightarrow \neg B) \wedge (B \leftrightarrow \neg C) \implies \neg A \leftrightarrow \neg C$	22	~ 0
4	$A \wedge B \to C, A \implies B \to C$	6	~ 0
5	$A \vee (B \wedge C) \implies (A \vee B) \wedge (A \vee C)$	11	~ 0
6	$A \vee B \vee C \vee D \vee E \vee F \vee G \vee H \implies$ A, B, C, D, E, F, G, H	15	~ 0
7	$s \doteq t \implies t \doteq s$	2	~ 0
8	$s \doteq t,\ t \doteq u,\ u \doteq v \implies s \doteq v$	4	~ 0
9	$s \doteq t,\ u \doteq v \implies f(s,u) \doteq f(t,v)$	3	~ 0
10	$g \doteq f(f(g)),\ h \doteq f(f(f(h))),$ $f(g) \doteq c,\ f(h) \doteq c \implies g \doteq h$	11	0.4s
11	$g \doteq f(f(g)),\ h \doteq f(f(f(h))),\ f(g) \doteq c,$ $f(h) \doteq c,\ P(g) \implies$ $P(h),\ P(f(f(f(f(f(f(f(f(f(c)))))))))$	9	21.5s
12	$\forall x\ R(x) \implies \exists x\ R(x)$	3	0.1s
13	$\forall x.\ P(x) \wedge Q(x) \implies \exists x\ \exists y.\ P(x) \wedge Q(y)$	4	0.1s
14	$\neg\forall x\ \forall y\ Q(x,y) \implies \exists y\ \exists x\ \neg Q(x,y)$	7	0.1s
15	$\exists x\ \forall y\ P(x,y) \implies \forall y\ \exists x\ P(x,y)$	5	0.1s
16	$\forall x.\ P(x) \to P(f(x)) \implies$ $\forall x.\ P(x) \to P(f(f(x)))$	10	0.1s
17	$P(f(c)),\ \forall x.\ f(x) \doteq x \implies P(f(f(f(c))))$	7	0.1s
18	$\exists x\ \exists y\ \neg x \doteq y,\ \forall x\ P(x) \implies \exists x.\ P(x) \wedge \neg x \doteq c$	18	1.1s
19	$\forall x\ P(x) \implies P(f(g(f(g(f(g(f(g(c)))))))))$	2	0.1s
20	$\forall x\ P(x) \implies P(f(g(\ \ldots^{20}\ f(g(c))\ \ldots^{20})))$	2	4.7s
21	$\forall x\ P(x) \implies P(f(g(h(i(j(k(c)))))))$	2	5.4s

1 The root sequents plus the number of subsequents used for the entire proof.

2 System: Linux Mint 18.1 64-bit, Linux-Kernel 4.8.0-34, Intel Core i7-2600 @ 3.4GHz x 4, 16GB Ram

increasing computation time when the term size grows. With some additional function symbols the Z3 Prover starts to exceed the timeout, that is why it becomes difficult for the Term Guessing Algorithms to find the right term. Especially for the last example the Z3 Prover fails multiple times – you may compare this example with the formula on page 53 –, while the Term Guessing Algorithm is able to recover from all theses fails by using the special handling for unknown results.

While the Term Guessing Algorithm is sound – every created proof is correct –, it is not complete because of two reasons: the success rate of the underlying SMT solver and a fragment of formulas that is currently unsolvable by the Term Guessing Algorithm.
The success of the implementation of the Term Guessing Algorithm in the Sequent Calculus Trainer highly depends on the return values of the Z3 Prover. If the Z3 Prover does not find solutions for a critical set of formulas, the Term Guessing Algorithm does not find a solution either. The lower the timeout used for the z3 Prover the higher the chance to exceed the timeout. This is why the Term Guessing Algorithm may fail more often with a lower timeout. The same applies to the contraction limit. If it is too low, the algorithm may not find needed instantiations. Therefore, it is very important to adjust the two options to fit the used input sequents.
The Partial Instantiation Algorithm is made to find a subset of terms that can be used at once to instantiate a quantified formula. But for sequents with hidden constants we perhaps need to create multiple instances of a quantified formula and instantiate only some of them until some other formulas of the sequent are resolved. For these sequents the instantiation of all instances of the quantified formula at once does not work. A sequent with a hidden constant contains left sided existential quantifiers or right sided universal quantifiers that are part of currently unreachable subformulas. These quantifiers would introduce new constant function symbols, but rule applications are not allowed on subformulas. The following example demonstrates this special type of sequents:

$$\forall x\, \exists y.\, P(x) \land Q(y) \implies \exists x\, \forall y.\, P(y) \land Q(x)$$

To solve this sequent we can either start with the left or right formula, but the universal quantifier on the left side cannot be instantiated until a new constant has been introduced by the instantiation of the right sided universal quantifier. Unfortunately, the right sided universal quantifier is currently hidden by an existential quantifier which cannot be resolved for the same reason. It depends on the new constant that is introduced by the left sided existential quantifier. As a result, both formulas depend on each other because of hidden constants. This dependency cycle can be resolved if the left formula is duplicated and one of the two instances is instantiated with a random term. Finally, the embedded existential quantifier can be instantiated with a new constant that is then used to break the dependency cycle.
Contractions on quantified formulas are normally used to do multiple ground term instantiations where each ground term is important to prove the sequent. In contrast, the sequent shown above needs an instantiation of the left sided quantified formula where the used ground term is completely exchangeable. The only reason for the instantiation

is to break the dependency cycle. These special type of sequents is currently unsolvable by the Term Guessing Algorithm.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P(a),\ Q(b),\ P(c),\ Q(d) \implies P(c)}(Ax) \quad \dfrac{}{P(a),\ Q(b),\ P(c),\ Q(d) \implies Q(b)}(Ax)}{P(a),\ Q(b),\ P(c),\ Q(d) \implies P(c) \wedge Q(b)}(\wedge_R)}{P(a),\ Q(b),\ P(c) \wedge Q(d) \implies P(c) \wedge Q(b)}(\wedge_L)}{P(a),\ Q(b),\ \exists y.\ P(c) \wedge Q(y) \implies P(c) \wedge Q(b)}(\exists_L)}{P(a),\ Q(b),\ \forall x\ \exists y.\ P(x) \wedge Q(y) \implies P(c) \wedge Q(b)}(\forall_L)}{P(a),\ Q(b),\ \forall x\ \exists y.\ P(x) \wedge Q(y) \implies \forall y.\ P(y) \wedge Q(b)}(\forall_R)}{P(a),\ Q(b),\ \forall x\ \exists y.\ P(x) \wedge Q(y) \implies \exists x\ \forall y.\ P(y) \wedge Q(x)}(\exists_R)}{P(a) \wedge Q(b),\ \forall x\ \exists y.\ P(x) \wedge Q(y) \implies \exists x\ \forall y.\ P(y) \wedge Q(x)}(\wedge_L)}{\exists y.\ P(a) \wedge Q(y),\ \forall x\ \exists y.\ P(x) \wedge Q(y) \implies \exists x\ \forall y.\ P(y) \wedge Q(x)}(\exists_L)}{\forall x\ \exists y.\ P(x) \wedge Q(y),\ \forall x\ \exists y.\ P(x) \wedge Q(y) \implies \exists x\ \forall y.\ P(y) \wedge Q(x)}(\forall_L)}{\forall x\ \exists y.\ P(x) \wedge Q(y) \implies \exists x\ \forall y.\ P(y) \wedge Q(x)}(Contr_L)$$

The failure for the small example on page 53 of the Z3 Prover, which is developed by a big team and over a long period of time with hundreds of thousands lines of code, shows how difficult it is to decide the validity problem for first-order formulas. For this reason, it was a good idea to build on top of existing solutions instead of creating everything new from scratch. Theorem provers like the Z3 Prover may be further improved in the future, which is why the Term Guessing Algorithm may also become better over the time.

As a conclusion of this thesis, we can state that we achieved the aim of making the Sequent Calculus Trainer smart. With the Term Guessing Algorithm the Sequent Calculus Trainer is now able to solve sequents or to give hints about how to solve a sequent. This is a very great feature that will be of tremendous benefit to all students who have to study the Sequent Calculus for the undergraduate degree.

List of Figures

List of Tables

Acknowledgements / Danksagung

An dieser Stelle möchte ich mich bei einigen Personen ganz herzlich bedanken, die einen wesentlichen Teil zum Gelingen dieser Masterarbeit beigetragen haben.
Ein ganz großes Dankeschön richtet sich an Professor Dr. Martin Lange und Dr. Norbert Hundeshagen für ihr außerordentliches Engagement auf fachlich höchstem Niveau und ihre Zeit, die sie für mich investiert haben. Für mich ist das Team um Professor Dr. Martin Lange einer der Hauptgründe, weshalb die Universität Kassel im Bezug auf das Informatikstudium anderen Universitäten gegenüber einen fachlichen und didaktischen Vorteil genießt. Vielen Dank dafür!
Ein weiterer Dank geht an meine Großmutter Waltraud Jacob und meine Mutter Barbara Jacob. Durch ihre Unterstützung ist mir das Studium der Informatik überhaupt erst ermöglicht worden.

References

[BCD+11] Clark Barrett et al. "CVC4". In: *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_14. URL: http://dx.doi.org/10.1007/978-3-642-22110-1_14.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. New York, NY, USA: Cambridge University Press, 1998. ISBN: 0-521-45520-0.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3-540-78799-2, 978-3-540-78799-0. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766.

[EFT07] Heinz-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Einführung in die mathematische Logik (5. Aufl.)* Spektrum Akademischer Verlag, 2007.

[EHL15] Arno Ehle, Norbert Hundeshagen, and Martin Lange. "The Sequent Calculus Trainer - Helping Students to Correctly Construct Proofs". In: *CoRR* abs/1507.03666 (2015). URL: http://arxiv.org/abs/1507.03666.

[Fit90] Melvin Fitting. *First-order Logic and Automated Theorem Proving*. Ed. by David Gries. New York, NY, USA: Springer-Verlag New York, Inc., 1990. ISBN: 0-387-97233-1.

[Gil60] P. C. Gilmore. "A Proof Method for Quantification Theory: Its Justification and Realization". In: *IBM J. Res. Dev.* 4.1 (Jan. 1960), pp. 28–35. ISSN: 0018-8646. DOI: 10.1147/rd.41.0028. URL: http://dx.doi.org/10.1147/rd.41.0028.

[LH17] Martin Lange and Norbert Hundeshagen. "Theoretische Informatik: Logik". Skript zur gleichnamigen Vorlesung. Universität Kassel 2017.

[RV02] Alexandre Riazanov and Andrei Voronkov. "The Design and Implementation of VAMPIRE". In: *AI Commun.* 15.2,3 (Aug. 2002), pp. 91–110. ISSN: 0921-7126. URL: http://dl.acm.org/citation.cfm?id=1218615.1218620.

[Sch13] Stephan Schulz. "System Description: E 1.8". In: *Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*. Ed. by Ken McMillan, Aart Middeldorp, and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 735–743. ISBN: 978-3-642-45221-5.

DOI: 10.1007/978-3-642-45221-5_49. URL: http://dx.doi.org/10.1007/978-3-642-45221-5_49.

[WDF+09] Christoph Weidenbach et al. "SPASS Version 3.5". In: *Automated Deduction - CADE-22 : 22nd International Conference on Automated Deduction*. Ed. by Renate A. Schmidt. Vol. 5663. Lecture Notes in Artificial Intelligence. Montreal, Canada: Springer, Aug. 2009, pp. 140–145. ISBN: 978-3-642-02958-5. DOI: 10.1007/978-3-642-02959-2_10.

Statement of authorship

I hereby certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. No other person's work has been used without due acknowledgement in this thesis. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged.

Place, Date Signature