DIPLOMARBEIT

Ein Kalkül für Beweisrepräsentationsstrukturen
(A calculus for Proof Representation Structures)

Angefertigt am
Mathematischen Institut

Vorgelegt der
Mathematisch-Naturwissenschaftlichen Fakultät der
Rheinischen Friedrich-Wilhelms-Universität Bonn

Januar 2009

Von

Daniel Kühlwein

Aus
Balingen

Acknowledgments

I would like to thank the whole Naproche team. Without them, this thesis would not have been possible. I am grateful to Bernhard Schröder for his comments and suggestions. Especially, I would like to thank my supervisor Peter Koepke for his support in the preparation of this thesis.

Last but not least, I want to express my gratitude to my family for their support, help, and advice throughout my studies.

Eidesstattliche Erklärung

Hiermit erkläre ich, Daniel Kühlwein, an Eides statt, dass ich die Diplomarbeit, "Ein Kalkül für Beweisrepräsentationsstrukturen", selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Bonn, den 3. November 2008

Unterschrift

Contents

Abstract

Was ist ein mathematischer Beweis? Ein mathematischer Beweis ist ein Text, der den Leser von der Richtigkeit einer Aussage überzeugen soll. Diese oder eine vergleichbare Aussage werden viele Personen, fachfremde und Mathematiker, als Antwort geben. Aber wie genau sieht solch ein "Beweistext" aus? Nach welchen Regeln ist er aufgebaut? Betrachtet man mathematische Beweise als eine Mischung von natürlicher Sprache und mathematischen Zeichen, wie man sie in mathematischen Lehrbüchern und Zeitschriften findet, so sind diese Fragen schwer zu beantworten. Abgesehen von solchen "normalen" Beweisen gibt es jedoch noch eine weitere Art von Beweisen: formale Beweise. Für diese ist es leicht eine Antwort zu finden: Das zugehörige Kalkül definiert, was erlaubt ist und was nicht. Alfred Whitehead und Bertrand Russell haben mit ihrem Werk *Principia Mathematica* [33] gezeigt, dass es theoretisch möglich ist, Mathematik rein formal zu betreiben. In den *Principia Mathematica* werden zwar nur die Grundlagen der Mathematik formalisiert, jedoch kann man daraus ableiten, dass es auch für den Rest möglich ist. Hierauf aufbauend kann man nun die zuvor genannten "normalen" Beweise als Abkürzungen für formale Beweise betrachten. Unter dieser Annahme wäre die einzige Regel für einen "normalen" Beweis, dass er eindeutig in einen formalen Beweis übersetzbar ist.

Inwieweit ist es möglich, "normale" Beweise (automatisch) in formale Beweise zu übersetzen? In welchem Ausmaß sind solche Übersetzungen eindeutig? Was ist die Sprache der Mathematik? Unter der Leitung von Peter Koepke (Mathematik, Universität Bonn) und Bernhard Schröder (Linguistik, Universität Duisburg-Essen) untersucht das Naproche-Projekt (NAtural language PROof CHEcking) diese und ähnliche Fragestellungen.

Als praktische Umsetzung dieser Überlegungen wird das Naproche-System, ein Computerprogramm, das "normale" Beweise auf logische Korrektheit überprüfen soll, entwickelt. In der aktuellen Version unterteilt sich das Naproche-System in drei Hauptmodule: Die Eingabeverarbeitung, die linguistische Verarbeitung und die logische Überprüfung. Ziel dieses Diplomprojektes war die Entwicklung und Implementierung des dritten Moduls, der logischen Überprüfung.

Nach der linguistischen Verarbeitung ist ein Eingabetext in eine so genannte Beweisrepräsentationsstruktur (Proof Representation Structure, PRS) transformiert worden. Die logische Überprüfung muss entscheiden, ob die dem Eingabetext entsprechende Beweisrepräsentationsstruktur logisch korrekt ist oder nicht. Man benötigt also ein Kalkül für Beweisrepräsentationsstrukturen. Diese Diplomarbeit beschreibt solch ein Kalkül. Sie unterteilt sich in fünf Kapitel:

Kapitel 1 gibt eine Einführung in diese Arbeit. Das Naproche Projekt wird beschrieben und für Naproche wichtige Arbeiten werden vorgestellt.

Kapitel 2 behandelt die Grundlagen der Logik erster Stufe. Zuerst werden eine Sprache erster Ordnung, sowie die dazugehörigen Terme und Formeln definiert. Danach wird die Semantik erklärt, und ein korrektes und vollständiges Sequenzenkalkül eingeführt.

In Kapitel 3 wird das Naproche-System ausführlicher behandelt. Das Kapitel beginnt mit der Beschreibung der groben Struktur des Naproche-Systems. Danach wird die momentan zulässige Eingabesprache definiert. Eine Einführung

in Diskursrepräsentationstheorie (Discourse Representation Theory, DRT), auf der die Linguistik des Naproche-Systems basiert, und Diskursrepräsentationsstrukturen (Discourse Representation Structures, DRS) wird gegeben. Daraufhin werden Beweisrepräsentationsstrukturen definiert. Automatische Beweiser (Automated Theorem Prover, ATP) und das TPTP-Projekt, welche für die logische Überprüfung der Beweisrepräsentationsstrukturen benutzt werden, werden vorgestellt und ihr Einsatz im Naproche-System erklärt. Das Kapitel schließt mit dem Burali-Forti-Paradoxon als einem Beispiel für ein nicht-triviales Theorem, das von dem aktuellen Naproche-System vollständig überprüft werden kann, ab.

Kapitel 4 definiert das Naproche-Kalkül für Beweisrepräsentationsstrukturen und zeigt dessen Vollständigkeit und Korrektheit: Nach einigen Lemmas wird zunächst das Formelbild (Formula Image) einer Beweisrepräsentationsstruktur definiert. Dies ist eine Abbildung, welche jeder Beweisrepräsentationsstruktur eine Sequenz von Formeln erster Stufe zuordnet. Danach wird das eigentliche Kalkül in Abhängigkeit von einem vorgegebenen Kalkül P definiert. Es folgt eine Betrachtung des Zusammenhangs zwischen der Eingabesprache des Naproche-Systems und Beweisrepräsentationsstrukturen, bevor das neu definierte Kalkül mit dem Quellcode des Naproche-Systems verglichen wird. Es wird gezeigt, dass die Ableitbarkeit einer Beweisrepräsentationsstruktur in dem Kalkül äquivalent dazu ist, dass das idealisierte Naproche-System die Beweisrepräsentationsstruktur akzeptiert. Zuletzt wird die Korrektheit und Vollständigkeit des Naproche-Kalküls gezeigt. Insbesondere werden folgende Theoreme bewiesen:

Theorem. Vollständigkeit des Naproche-Kalküls
Ist P ein vollständiges Kalkül, dann ist das Naproche-Kalkül vollständig. D.h. falls $\Gamma \models \varphi$, dann ist die Beweisrepräsentationsstruktur, die nur die Annahmen-Bedingung mit der Annahme $\bigwedge \Gamma$ und der Folgerung φ enthält, Naproche-akzeptiert.

Theorem. Korrektheit des Naproche-Kalküls
Ist P ein korrektes Kalkül, dann ist das Naproche-Kalkül korrekt. D.h. falls A eine Naproche-akzeptierte Beweisrepräsentationsstruktur ohne Definitions-Bedingungen ist, dann ist das Formelbild $FI(A)$ in dem Sequenzenkalkül ableitbar.

Der Beweis erfolgt durch eine doppelte Induktion über die Tiefe der Beweisrepräsentationsstruktur und die Anzahl ihrer Bedingungen. Im Induktionsschritt werden die vorherigen Definitionen, das Kalkül für Beweisrepräsentationsstrukturen und deren Formelbild, zusammen mit dem Sequenzenkalkül aus Kapitel 2 benutzt.

Im letzten Kapitel werden zunächst einige Probleme des Naproche-Systems aufgezeigt. Der Einfluss des automatischen Beweisers und der Eingabesprache wird näher betrachtet. Danach werden mögliche Verbesserungen an dem aktuellen Programm besprochen. Insbesondere werden Ideen zur Verbesserung der Benutzerfreundlichkeit und der Beweisfähigkeit genannt. Zuletzt wird kurz die geplante zukünftige Zusammenarbeit mit dem VeriMathDoc Projekt in Saarbrücken und Bremen beschrieben.

Chapter 1

Introduction

Considering that mathematics has a reputation of being an exact science, it is interesting to note that one of the main concepts of mathematics, the mathematical proof, is somewhat vaguely defined. What exactly is a mathematical proof? Firstly, one can distinguish between two kinds of mathematical proofs: Formal and informal proofs.

An informal proof is a mixture of natural language and mathematical symbols. Most proofs in mathematical textbooks and journals are informal. Informal proofs are subjective. People may disagree about whether a text is an informal proof or not.

Formal proofs are finite derivations in a calculus. They are sequences of mathematical symbols and do not contain natural language elements. Given a calculus, there is a definite answer of whether or not a text is a formal proof. In the *Principia Mathematica* [33], Alfred Whitehead and Bertrand Russell formally proved many theorems which were considered the foundation of mathematics. Since then, most mathematicians agree that it would be possible, even if extremely tedious, to find a formal proof for every theorem. With this in mind, one could interpret informal proofs as abbreviations for formal proofs.

1.1 The Naproche Project

Naproche, short for NAtural language PROof CHEcking, is a project that was founded by Bernhard Schröder (Linguistics, University of Duisburg-Essen) and Peter Koepke (Mathematical Institute, University of Bonn) to study informal mathematical proofs. Among the questions we ask ourselves are: "Is it possible to (automatically) translate an informal into a formal proof?", "Can such a translation be unambiguous ?", and "What is the syntax and semantics of informal proofs?".

As an answer to these questions, we develop the Naproche system, a computer program that aims to check proofs that are written in a controlled natural language, the Naproche language, for correctness. Even though the Naproche system and the Naproche language are quite new, we could already get some results. The Naproche language is strong enough to formulate proofs for non-trivial theorems like the Burali-Forti paradox (See 3.4), and the proof from section 3.4 can be checked successfully by the Naproche system. Eventually, we hope that the Naproche system can be used to automatically create translations from informal into formal proofs.

1.2 This Thesis

In its current version, the Naproche system is divided into three main modules: the input module, the linguistic module, and the logic module. The goal of this diploma project was the development and implementation* of the logic module.

More specifically, the first two modules transform the input text into a Proof Representation Structure. The logic module must decide if the Proof Representation Structure is correct. Thus, the logic module is the implementation of a calculus for Proof Representation Structures. This thesis describes the calculus and shows its completeness and correctness.

The structure of the thesis is as follows: Chapter 2 introduces the notions and definitions of first order logic which will be used in the remainder of the text. The Naproche system is then described in greater detail in chapter 3. Chapter 4 defines the calculus for Proof Representation Structures, and shows its correctness and completeness. The final chapter takes a look at the limitations of the concept of the Naproche system, shows some problems which were found during the development of the last version, and presents the plan for the near future of Naproche.

1.3 Related Work

To the authors knowledge, there are two groups which are very closely related to Naproche. The VeriMathDoc project in Saarbrücken [31], [32], and the SAD team in Kyiv [15]. They all share the concept of a controlled natural language as input, linguistic parsing and automated proof checking.

Claus Zinn wrote his doctoral thesis [34] about the natural language of mathematics, but unfortunately his ideas were not developed any further.

The probably historically most prominent representative of automated proof verification is the Mizar system [16]. It was started in 1973 by Andrzej Try-

* The implementation was done between April and June 2008 by Dörthe Arndt and the author, with additional help of the trainees Bhoomija Ranjan and Shruti Gupta.

BULEC, and since then an impressive library of mathematical texts written in the Mizar language, as well as a journal, Formalized Mathematics, in which these articles are published, were created.

Automated theorem provers are used in the checking process of the Naproche-System. The best starting point for information in this area is arguably the TPTP homepage [23]. We should also mention the proof assistants COQ [5] and Isabelle [18].

On the linguistic side, the Naproche system is based upon Discourse Representation Theory, in particular the work of Hans Kamp and Uwe Reyle [10]. For our needs, we extended the original Discourse Representation Theory and defined Proof Representation Structures. Nickolay Kolev wrote his Magister thesis about this topic [13].

Norbert Fuchs created Attempto Controlled English [21], a controlled natural language which reads like normal English and has an unambiguous translation into first order logic.

Chapter 2

Terms, Formulas and Calculi

This chapter briefly introduces the basic notions and definitions of first order logic which will be used in the remainder of this thesis. We take a look at the syntax and semantics of first order languages, and define a complete and correct sequent calculus.

If you are not already familiar with first order logic, then you might want to pick up an introductory logic book for additional and more detailed information. This chapter is based upon Ebbinghaus' textbook [3] and Peter Koepke's lecture notes [11]. Most importantly, the sequent calculus is taken from [11].

2.1 First Order Language

We give a brief recapitulation of the basic definitions of a first order language.

Definition 2.1.1. *Alphabet and Words over an Alphabet*
An alphabet A is a non-empty set of symbols. A finite sequence of elements of A is called a word of the alphabet A. A^* is defined as the set of all words over an alphabet A.

Definition 2.1.2. *Alphabet of a First Order Language*
The alphabet of a first order language L contains the following symbols:

- v_0,v_1,.. , a countably infinite number of variables
- $\neg, \rightarrow, \bot$, logical symbols with the usual meaning
- $\forall$, universal quantifier
- $\equiv$, equality
-), (, brackets

- for each $n \in \mathbb{N}$ a possibly empty set of n-ary relation symbols
- for each $n \in \mathbb{N}$ a possibly empty set of n-ary function symbols
- a possibly empty set of constant symbols

Definition 2.1.3. *Terms of a First Order Language*
Let A be an alphabet of a first order language L. The set of terms of L is the minimal subset T of A^* for which the following holds:

- Each variable is an element of T.
- Each constant of A is an element of T.
- If $t_1, .., t_n$ are elements of T, and f is an n-ary function symbol in A, then $ft_1..t_n$ is an element of T.

The elements of T are called L-terms.

Definition 2.1.4. *First Order Formulas*
Let A be an alphabet of a first order language L. The set of formulas of L is the minimal subset F of A^* for which the following holds:

- $\perp$ is an element of F.
- For L-terms t_1, t_2, $(t_1 \equiv t_2)$ is an element of F.
- Let $t_1, ..t_n$ be L-terms and let R be an n-ary relation symbol in A. Then $Rt_1..t_n$ is an element of F.
- If φ is an element of F, then $\neg\varphi$ is an element of F.
- If φ and ψ are elements of F, then $(\varphi \to \psi)$ is an element of F.
- If φ is an element of F, and x is a variable, then $\forall x\varphi$ is an element of F.

The elements of F are called L-formulas.

Definition 2.1.5. *Variables of a Term*
Let L be a first order language. Let x be a variable of L, c be a constant of L, and $t_1, .., t_n$ be L-terms. We define the function *var* for terms:

$$\begin{aligned} var(x) &= \{x\} \\ var(c) &= \emptyset \\ var(ft_1, .., t_n) &= var(t_1) \cup .. \cup var(t_n) \end{aligned}$$

Definition 2.1.6. *Free Variables of a Formula*
Let L be a first order language. We define the free variables of an L-formula inductively:

$$\begin{aligned} free(\bot) &= \emptyset \\ free((t_1 = t_2)) &= var(t_1) \cup var(t_2) \\ free(Rt_1...t_n) &= var(t_1) \cup ... \cup var(t_n) \\ free(\neg\varphi) &= free(\varphi) \\ free((\varphi \to \psi)) &= free(\varphi) \cup free(\psi) \\ free(\forall x\varphi) &= free(\varphi)\backslash\{x\} \end{aligned}$$

Definition 2.1.7. *Substitution in a Term*
Let L be a first order language, t, u be L-terms, and x be a variable of L. We define the substitution $t\frac{u}{x}$ of u for x recursively:

- For all variables $y \neq x$, $y\frac{u}{x} = y$.
- $x\frac{u}{x} = u$.
- $c\frac{u}{x} = c$ for all constants c.
- $(ft_1..t_n)\frac{u}{x} = ft_1\frac{u}{x}...t_n\frac{u}{x}$ for all n-ary function symbols f.

Definition 2.1.8. *Substitution in a Formula*
Let L be a first order language, φ be an L-formula, x be a variable of L and t be an L-term. We define the substitution $\varphi\frac{t}{x}$ of t for x recursively:

- $\bot\frac{t}{x} = \bot$.
- For all L-terms t_1, t_2, $(t_1 \equiv t_2)\frac{t}{x} = (t_1\frac{t}{x} \equiv t_2\frac{t}{x})$.
- For all L-terms $t_1, ..t_n$ and n-ary relation symbols R, $(Rt_1..t_n)\frac{t}{x} = Rt_1\frac{t}{x}...t_n\frac{t}{x}$.
- For all L-formulas ψ, $(\neg\psi)\frac{t}{x} = \neg(\psi\frac{t}{x})$.
- For all L-formulas ψ_1, ψ_2, $(\psi_1 \to \psi_2)\frac{t}{x} = (\psi_1\frac{t}{x} \to \psi_2\frac{t}{x})$
- For all L-formulas ψ, if $x = y$, then $(\forall y\psi)\frac{t}{x} = \forall y\psi$.
- For all L-formulas ψ, if $x \neq y$, then let u be a variable which does not occur in $\forall y\psi, x$ and t. We define $(\forall y\psi)\frac{t}{x} = \forall u((\psi\frac{u}{y})\frac{t}{x})$.

The reader might miss the logical symbols $\top, \wedge, \vee, \leftrightarrow$ and $\exists$, which can often be found in textbooks. We will use them as syntactic sugar in our formulas. Their use can greatly increase the readability of longer formulas.

Definition 2.1.9. *Abbreviated Formulas*
We define the abbreviations

- $\top$ for $\neg\bot$.
- $\varphi \vee \psi$ for $\neg\varphi \rightarrow \psi$.
- $\varphi \wedge \psi$ for $\neg(\varphi \rightarrow \neg\psi)$.
- $\varphi \leftrightarrow \psi$ for $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$.
- $\exists x\, \varphi$ for $\neg\forall x\, \neg\varphi$.

2.2 The Semantics of First Order Languages

By themselves, first order formulas are just a special kind of strings of symbols, without any particular meaning. In order to make any sense of a formula, we need to interpret it in a model. We define interpretations of first order languages as well as the relation $\models$ between an interpretation and a formula and between a set of formulas and a formula.

Definition 2.2.1. *Interpretation*
Let L be a first order language. An L-interpretation is a tupel (A, β) such that

- A is a non-empty set
- for each n-ary relation symbol R of L, $\beta(R)$ is an n-ary relation on A.
- for each n-ary function symbol F of L, $\beta(F)$ is an n-ary function on A.
- for each constant c of L is $\beta(c)$ an element of A.
- for each variable v_i of L is $\beta(v_i)$ an element of A.

We say that L is interpreted in the model A via the map β.

Definition 2.2.2. Let $I = (A, \beta)$ be an L-interpretation.

- For each variable x let $I(x) = \beta(x)$.
- For each constant c let $I(c) = \beta(c)$.
- For each n-ary function symbol f of L and terms $t_1, ..t_n$ let $I(ft_1...t_n) = \beta(f)(I(t_1), ..I(t_n))$.

For a variable x of L and an element a of A we define the L-interpretation $I\frac{a}{x} = (A, \beta')$, where $\beta'(x) = a$ and for all $y \neq x$ $\beta'(y) = \beta(y)$.

Definition 2.2.3. *The models relation* $\models$
Let $I = (A, \beta)$ be an L-interpretation, $t_1, .., t_n$ be L-terms and φ,ψ be L-formulas. We define:

- $I \models \bot$ is always false.
- $I \models t_1 \equiv t_2$ iff: $I(t_1) = I(t_2)$.
- $I \models Rt_1...t_n$ iff: $\beta(R)(I(t_1), ..., I(t_n))$.
- $I \models \neg\varphi$ iff: not $I \models \varphi$.
- $I \models \varphi \to \psi$ iff: not $I \models \varphi$, or $I \models \psi$.
- $I \models \forall x\, \varphi$ iff: for all $a \in A$ $I\frac{a}{x} \models \varphi$.

When we consider the abbreviated formulas from definition 2.1.9 under an interpretation, we get the expected behaviour:

Lemma 2.2.1. *Let L be a first order language, $I = (A, \beta)$ be an L-interpretation, φ, ψ be L-formulas. Then the following holds:*

- $I \models \top$ *is always true.*
- $I \models \varphi \wedge \psi$ *iff:* $I \models \varphi$ *and* $I \models \psi$.
- $I \models \varphi \vee \psi$ *iff:* $I \models \varphi$ *or* $I \models \psi$.
- $I \models \varphi \leftrightarrow \psi$ *iff:* $I \models \varphi$ *if and only if* $I \models \psi$.
- $I \models \exists x\, \varphi$ *iff: there exists* $a \in A$ *such that* $I\frac{a}{x} \models \varphi$.

Proof: Obvious.

Definition 2.2.4. Let L be a first order language, let Γ be a set of L-formulas, I be an L-interpretation. We write $I \models \Gamma$, if $I \models \varphi$ for all $\varphi \in \Gamma$.

Definition 2.2.5. $\Gamma \models \varphi$
Let L be a first order language. let Γ be a set of L-formulas, and φ be an L-formula. We say that $\Gamma \models \varphi$, if and only if, for all L-interpretations I, if $I \models \Gamma$ then $I \models \varphi$.

2.3 A Sequent Calculus

After defining the semantics of a formula, the question of figuring out whether for a set of formulas Γ and a formula φ holds $\Gamma \models \varphi$ becomes eminent. In order to answer this, formula calculi were developed. A formula calculus is a set of rules which allows to form new formulas from existing formulas. In this section, we introduce a calculus which is complete and correct: $\Gamma \models \varphi$ if and only if $\Gamma\ \varphi$ can be derived in the calculus. Note that the calculus definition is taken from Peter Koepke's logic lecture [11].

Fix a language L for this section.

Definition 2.3.1. *Sequents*
A finite sequence of formulas $\langle \varphi_1, ...\varphi_n, \varphi_{n+1} \rangle$ is called a sequent. The initial segment of a sequent $\langle \varphi_1, ...\varphi_n \rangle$ is denoted by Γ. We also write $\Gamma\varphi$ for the sequent $\langle \varphi_1, ...\varphi_n, \varphi \rangle$

Definition 2.3.2. *Free Variables of a Sequent*
For a sequent $\langle \varphi_1, ...\varphi_n \rangle$, we define $free(\langle \varphi_1, ...\varphi_n \rangle) = \bigcup_{i=1}^{n} free(\varphi_i)$.

Definition 2.3.3. *A Sequent Calculus*

We write down the rules of our calculus in the form $\dfrac{Premises}{Result}$, meaning that if we already have *Premises*, then we can use this rule to get *Result*. Our sequent calculus has the following rules:

- Assumption $\dfrac{}{\Gamma \quad \varphi \quad \varphi}$
- Monotonicity $\dfrac{\Gamma \quad \varphi}{\Gamma \quad \psi \quad \varphi}$
- $\rightarrow$ Introduction $\dfrac{\Gamma \quad \varphi \qquad \psi}{\Gamma \quad \varphi \rightarrow \psi}$
- $\rightarrow$ Elimination $\dfrac{\begin{array}{l}\Gamma \quad \varphi \\ \Gamma \quad \varphi \rightarrow \psi\end{array}}{\Gamma \quad \psi}$
- $\bot$ Introduction $\dfrac{\begin{array}{l}\Gamma \quad \varphi \\ \Gamma \quad \neg\varphi\end{array}}{\Gamma \quad \bot}$
- $\bot$ Elimination $\dfrac{\Gamma \quad \neg\varphi \quad \bot}{\Gamma \quad \varphi}$
- $\forall$ Introduction $\dfrac{\Gamma \quad \varphi\frac{y}{x}}{\Gamma \quad \forall x\, \varphi}$ if $y \notin free(\Gamma\ \forall x\varphi)$
- $\forall$ Elimination $\dfrac{\Gamma \quad \forall x\, \varphi}{\Gamma \quad \varphi\frac{t}{x}}$ for all terms t.
- $\equiv$ Introduction $\dfrac{}{t \equiv t}$ for all terms t.

- $\equiv$ Elimination
$$\frac{\begin{array}{ll}\Gamma & \varphi\frac{t}{x}\\ \Gamma & t \equiv t'\end{array}}{\Gamma \quad \varphi\frac{t'}{x}}$$

Definition 2.3.4. $\vdash$
A sequent $\Gamma\ \varphi$ which can be created by finitely many applications of the rules in a calculus is called derivable from that calculus. We write $\Gamma \vdash \varphi$ if $\Gamma\ \varphi$ can be derived from the calculus.

Remark. Assume that $\Gamma \vdash \varphi$. Then it can be shown that only the set of the formulas in Γ is relevant for the derivation. See chapter 11.3 in [11] for details.

Definition 2.3.5. *Correctness and Completeness*
A calculus is called correct, if for all sequents $\Gamma \vdash \varphi$ implies $\Gamma \models \varphi$.
A calculus is called complete, if for all sequents $\Gamma \models \varphi$ implies $\Gamma \vdash \varphi$.

Theorem 2.3.1. *This sequent calculus is complete and correct.*

> **Proof:** Historically, GÖDEL was the first to prove the completeness of a calculus [7]. The proof for this calculus is based on HENKIN's work [8] and can be found in KOEPKE's lectures notes [11].

We will now derive four additional rules for the sequent calculus. Derivations in the calculus will be annotated with the line number and additional information about the rule which was used, as well as the premises which were used for the rule. We abbreviate Monotonicity "Mono.", Assumption "Assump.", Introduction "Intro.", and Elimination "Ele.". The premises of the rule we use are written in brackets before the name of the rule. For example, consider the following line:

$$\Gamma \quad \neg\varphi \quad \bot \qquad (2,3) + \bot \text{ Intro.} \quad (4)$$

The line number is (4). We used line (2) and (3) as premises for the $\bot$ Introduction rule to derive the sequent $\Gamma\ \neg\varphi\ \bot$.

Lemma 2.3.2. *Assume that we derived* $\Gamma\ \neg\neg\varphi$ *in the sequent calculus. Then we can derive* $\Gamma\ \varphi$.

Proof:

Γ	$\neg\neg\varphi$			(1)
Γ	$\neg\varphi$	$\neg\varphi$	Assump.	(2)
Γ	$\neg\varphi$	$\neg\neg\varphi$	(1) + Mono.	(3)
Γ	$\neg\varphi$	$\bot$	(2,3) + $\bot$ Intro.	(4)
Γ	φ		(4) + $\bot$ Ele.	(5)

Lemma 2.3.3. *For every sequent* Γ, *we can derive* $\Gamma\ \top$ *in the sequent calculus.*

Proof:

Γ	$\neg\neg\bot \quad \neg\neg\bot$	Assump.	(1)
Γ	$\neg\neg\bot \quad \bot$	(1) + Lemma 2.3.2	(2)
Γ	$\top$	(2) + $\bot$ Ele.	(3)

Lemma 2.3.4. *Assume that we derived $\Gamma\ \neg(\varphi \to \neg\psi)$ in the sequent calculus. Then we can derive $\Gamma\ \varphi$. Because we defined $\varphi \wedge \psi$ as $\neg(\varphi \to \neg\psi)$, this lemma can be rephrased to: The derivability of $\Gamma\ \varphi \wedge \psi$ in the sequent calculus implies the derivability of $\Gamma\ \varphi$ in the sequent calculus.*

Proof:

Γ	$\neg(\varphi \to \neg\psi)$				(1)
Γ	$\neg\varphi$	$\neg\varphi$		Assump.	(2)
Γ	$\neg\varphi$	φ	φ	Assump.	(3)
Γ	$\neg\varphi$	φ	$\neg\varphi$	(2) + Mono.	(4)
Γ	$\neg\varphi$	φ	$\bot$	(3,4) + $\bot$ Intro.	(5)
Γ	$\neg\varphi$	φ	$\neg\neg\psi \quad \bot$	(5) + Mono.	(6)
Γ	$\neg\varphi$	φ	$\neg\psi$	(6) + $\bot$ Ele.	(7)
Γ	$\neg\varphi$	$\varphi \to \neg\psi$		(7) + $\to$ Intro.	(8)
Γ	$\neg\varphi$	$\neg(\varphi \to \neg\psi)$		(1) + Mono.	(9)
Γ	$\neg\varphi$	$\bot$		(8,9) + $\bot$ Intro.	(10)
Γ	φ			(10) + $\bot$ Ele.	(11)

Lemma 2.3.5. *Assume that we derived $\Gamma\ \neg(\varphi \to \neg\psi)$ in the sequent calculus. Then we can derive $\Gamma\ \psi$. Because we defined $\varphi \wedge \psi$ as $\neg(\varphi \to \neg\psi)$, this lemma can be rephrased to: The derivability of $\Gamma\ \varphi \wedge \psi$ in the sequent calculus implies the derivability of $\Gamma\ \psi$ in the sequent calculus.*

Proof:

Γ	$\neg(\varphi \to \neg\psi)$			(1)
Γ	$\neg\psi$	$\neg\psi$	Assump.	(2)
Γ	$\neg\psi$	$\varphi \to \neg\psi$	(2) + Mono. + $\to$ Intro.	(3)
Γ	$\neg\psi$	$\bot$	(1,3) + $\bot$ Intro.	(4)
Γ	ψ		(4) + $\bot$ Ele.	(5)

So we can add the following additional rules to our calculus:

$$\frac{\Gamma \quad \neg\neg\varphi}{\Gamma \quad \varphi}$$

$$\frac{}{\Gamma \quad \top}$$

$$\frac{\Gamma \quad \neg(\varphi \to \neg\psi)}{\Gamma \quad \varphi}$$

$$\frac{\Gamma \quad \neg(\varphi \to \neg\psi)}{\Gamma \quad \psi}$$

Chapter 3

The Naproche System

The Naproche system is a computer program which checks texts that are written in a controlled natural language, the Naproche language, for correctness. Ultimately, we would like the Naproche system to be able to check realistic natural language mathematical texts, similar to those that can be found in journals and textbooks, and to provide automated translations from such informal into formal proofs.

This chapter gives an overview of the general design of the current Naproche system. It introduces the theories which form the foundation of the implementation, as well as the software that is used. In section 3.1 we explain the layout of the Naproche system, in particular its division into the input module, the linguistic module, and the logic module. We then take a closer look at the linguistic part of the Naproche system (3.2). The Naproche language is defined (3.2.1), and a brief introduction to Discourse Representation Theory is given (3.2.2). In the remainder of the section, we give the definitions for Proof Representation Structures (3.2.3) and show how they are created (3.2.4). Section 3.3 gives a short introduction to automated theorem provers and the TPTP project. Finally, section 3.4 shows a proof of the Burali-Forti paradox, written in the Naproche language, as an example for a mathematical text which can be checked by the Naproche system.

We abbreviate automated theorem prover with ATP, Discourse Representation Theory with DRT, Discourse Representation Structure with DRS, and Proof Representation Structure with PRS.

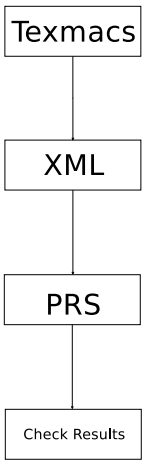

Figure 3.1: The structure of the Naproche system

3.1 An Overview

The first version of the Naproche system was written by Peter Koepke. The input text was not processed linguistically, and an internal calculus was used for the verification of proof steps. Because of the obvious limitations, the Naproche system was redesigned in 2007.

In its current version, the Naproche system is devised as a plugin for the WYSIWYG editor Texmacs [4]. If one wants to check a text with the Naproche system, one simply has to press the appropriate button. The input then gets processed by Naproche, giving *Proof Accepted* or *Proof NOT accepted* as output.

The program consists of three modules: Firstly, the input module which creates an XML file from the Texmacs input and reports the result of the check back to the user. Secondly, the linguistic module which takes such an XML file and produces the corresponding PRS. Thirdly, the logic module which checks the PRS for correctness. Figure 3.1 gives a graphical representation of the structure of the Naproche system.

We will now consider the current versions of the modules in greater detail:

The input module: Michael Klein wrote the first version of this module. Since then it has been extended by Friedemann Koepke and Nickolay Kolev. The translation process creates an XML file, which contains the original input annotated with unique labels for each sentence [13]. This XML file is further processed by the rest of the Naproche system, and the result is reported as either *Proof Accepted* or *Proof NOT accepted*. Internally, the Naproche system already produces a much more informative output, making it easy to understand what is going on underneath, and potentially giving a much better feedback, especially in the case that a proof cannot be checked by the Naproche system.

Under the supervision of Gregor Büchel from the Cologne University of

Applied Sciences, Sebastian Zittermann is working on a better Texmacs interaction, so that in future versions we can have more information in the XML file, and a more detailed report of the checking process.

The linguistic module: Nickolay Kolev's Magister project [13] was to devise and implement this module. The definitions of a PRS are given in 3.2.3, and the constructions algorithm is explained by example in 3.2.4.

The logic module was implemented by the author with additional help from Dörthe Arndt, Bhoomija Ranjan and Shruti Gupta. The PRS is, step by step, translated into first order logic, and, if necessary, checked by an ATP. The details of the algorithm are described in chapter 4. For the interaction with the ATP, we use software from the TPTP project, which is described in 3.3.

Note that the output *Proof NOT accepted* does not imply that the proof is wrong! It only means that Naproche was not able to verify at least one step in the proof.

3.2 The Linguistics of the Naproche System

The linguistic challenge of the Naproche system is to teach the computer how to "understand" the input text. Computer linguistics has several approaches for this problem. We chose one of these, Discourse Representation Theory, as linguistic framework for the Naproche system. Discourse Representation Theory offers Discourse Representation Structures as a method to extract the meaning of a text. Since mathematical texts have several peculiarities in comparison to natural language texts, we had to change this approach a bit, and, as a result, developed Proof Representation Structures [13].

We will first define the controlled natural language which the Naproche system currently accepts. Then, we take a look at Discourse Representation Theory and Discourse Representation Structures, and lastly we explain Proof Representation Structures as they are used in the Naproche system.

Hans Kamp developed DRT and DRS. The book *From Discourse to Logic* that he wrote together with Uwe Reyle gives a good and thorough introduction. Note that, even though Zinn [34] also uses the notion of a PRS, and has similar ideas, his definition of a PRS is not the one we will use in the sequel.

For the rest of the chapter, we use *discourse* synonymously with text.

3.2.1 The Naproche Language

The Naproche language is the controlled natural language which can be parsed and processed by the current Naproche system. It was designed to handle the peculiarities of mathematical texts and even though it is still a very limited language, it is interesting to note that texts, which are written with these limited

means, do already look quite similar to real mathematical text. We will see an example for this in section 3.4.

In the future, we would like to expand the Naproche language. At best, we would like the Naproche language to be so natural and expressive, that one can write Naproche texts which a human reader cannot distinguish from a natural language mathematical text. The original definition of the Naproche language can be found in [13].

Definition 3.2.1. *The Naproche Language*
The current version of the Naproche language consists of

- *Structure markers*
  Theorem, *Lemma*, *Proof*, *Qed*.

- *Statements*

  ```
  statement --> (statement_trigger), sub-statement.
  ```

  Where statement_trigger is either empty or one of the following: *then*, *hence*, *recall that*, *but*, *in particular*, *observe that*, *together we have* and *so*.

- *Definitions*

  ```
  define --> "define", sub-statement, "if and only if",
     sub-statement.
  define --> "define", sub-statement, "iff", sub-statement.
  ```

- *Assumptions*

  ```
  assumption --> (assumption_trigger), sub-statement.
  ```

  Where assumption_trigger is one of the following: *let*, *consider*, *assume that* and *assume for a contradiction that*.

- *Assumption closing*

  ```
  assumption-close --> "thus", sub-statement.
  ```

- *Quantification and implies*

  ```
  quantification --> "for all", variable, ",",
     sub-statement.
  quantification --> "there is an", variable, "such that",
     sub-statement.
  implication --> sub-statement, "implies", sub-statement
  ```

- *Negation*

```
negation --> "not", sub-statement.
```

In all these cases, *sub-statement* is a formula that is created from a fixed underlying first order language, and *variable* is a variable from the same language. The complete grammar of the current Naproche language in Backus-Naur Form can be found in appendix C, and the first order language which is used in the Naproche system is defined in appendix B.

3.2.2 Discourse Representation Theory and Discourse Representation Structures

What exactly is Discourse Representation Theory?

> *DRT offers a natural architecture for thinking about the way information is accumulated in the course of discourse processing: essentially it allows us to draw pictures of the changing context. [1]*

The main idea behind DRT is to see a discourse in its context. For example, when we consider the sentence *"She ate the cake"*, then, in order to figure out who is meant with *she*, we have to look at the information in the sentences preceding this sentence. The entities which we encountered in the text so far are part of the context. They are also called discourse referents. Basic DRT stops here, but there are modifications where the meaning of context is extended. For example the tense in which the text is written could be part of the context.

In DRT, the discourse is processed incrementally. Each sentence is first interpreted in the context so far and then the context gets updated according to the content of the processed sentence.

We will present the simplest version of a DRS, in which context only means accessible discourse referents, by giving the basic definitions, taken from [1], as well as two small examples.

Definition 3.2.2. *DRS and DRS conditions*
Let L be a first order language. We define DRSs and DRS conditions by a simultaneous recursion. Let $x_1, ..., x_n$ be variables of L (also called discourse referents), and $\gamma_1, ...\gamma_m$ be DRS conditions, then $D = (\langle x_1, ..., x_n\rangle, \langle\gamma_1, ..., \gamma_m\rangle)$ is a DRS. DRS conditions are defined as follows:

- If R is an n-ary relation symbol of L, and $t_1, ..., t_n$ are terms of L, then $R(t_1, ..., t_n)$ is a condition.
- If t_1, t_2 are terms of L, then $t_1 \equiv t_2$ is a condition.
- If B is a DRS, then $\neg B$ is a condition.

- If B_1 and B_2 are DRS, then $B_1 \vee B_2$ is a condition.
- If B_1 and B_2 are DRS, then $B_1 \Rightarrow B_2$ is a condition.
- Nothing else is a condition.

Let R be an n-ary relation symbol of L, and $t_1, ..., t_n$ be terms of L, then a DRS condition of the form $R(t_1, ..., t_n)$ is called a relation condition, and a DRS condition of the form $t_1 \equiv t_2$ is called an equivalence condition.

There is also a graphical representation of a DRS, which is, due to readability, more commonly used. In the graphical notation, a DRS $D = (\langle x_1, ..., x_n \rangle, \langle \gamma_1, ..., \gamma_m \rangle)$ is written as:

$x_1, ...x_n$
γ_1
$\vdots$
γ_n

To keep track of the context of a DRS, we define accessibility. Roughly speaking, everything which is accessible from a DRS is part of the context of that DRS.

Definition 3.2.3. *Accessibility*
A DRS B is accessibly from a DRS C if, and only if, $B = C$ or B *subordinates* C.

Definition 3.2.4. *Subordinates*
A DRS B subordinates a DRS C if, and only if B immediately subordinates C, or if there is some DRS D such that B immediately subordinates D and D subordinates C.

Definition 3.2.5. *Immediately subordinates*
A DRS B immediately subordinates a DRS C if, and only if

- B contains a condition of the form $\neg C$.
- B contains a condition of the form $C \vee D$ or $D \vee C$ for some DRS D.
- B contains a condition of the form $C \Rightarrow D$ for some DRS D.
- $B \Rightarrow C$ is a condition in some DRS D.

When we consider the graphical representation of a DRS, this basically means that every piece of information which stands above, or on the left, of a DRS D is accessible from D.

We will now consider a few examples, namely we will take a look at two discourses and show their corresponding DRS. If you are interested in the details, chapter 1 from [1] gives a good introduction.

Example 3.2.1. The sentence *"Peter is big."* has the corresponding DRS:

x
$x \equiv PETER$ $BIG(x)$

Example 3.2.2. The discourse *"Peter is big. He lives in New York."* has the corresponding DRS:

x, y, z
$x \equiv PETER$ $BIG(x)$ $y \equiv NEW\ YORK$ $LIVE(z, y)$ $x \equiv z$

This example shows how accessibility is used for pronoun resolution. In order to figure out who is meant by the pronoun *"he"* in the second sentence, we consider the discourse referents which are accessible in the current DRS. In this simple example, the only accessible discourse referent we have is x. Therefore, *"he"* must refer to x and we can add $x \equiv z$ to our conditions.

Now, x is also the only discourse referent which is available at that point, so one might wonder if accessibility is really necessary. Imagine a bigger text where we talk about two men, *Peter* and *Paul*. If in such a text the pronoun *"he"* appears, it is not be obvious to whom it refers to. In this a case, accessibility helps, because it restricts the available discourse referents.

The Semantic of DRS

Having defined DRS, we can now interpret DRS in different models. We define the truth of a DRS with respect to an interpretation. For additional reading, take a look at [30].

Definition 3.2.6. *Assignments and partial Interpretations*
Let L be a first order language. A partial L-interpretation is a tupel (A, β) such that

- A is a non-empty set
- for each n-ary relation symbol R of L, $\beta(R)$ is an n-ary relation on A.

- for each n-ary function symbol F of L, $\beta(F)$ is an n-ary function on A.
- for each constant c of L is $\beta(c)$ an element of A.

Given a partial L-interpretation (A, β), an assignment f is a map from a subset of the variables of L to A.

Definition 3.2.7. Let (A, β) be a partial L-interpretation and f be an assignment. Then we define the map α_f:

- For each constant c let $\alpha_f(c) = \beta(c)$
- For each variable x for which $f(x)$ is defined let $\alpha_f(x) = f(x)$
- For each n-ary function symbol g of L and L-terms $t_1, ...t_n$ which only contain variables for which f is defined, let $\alpha_f(gt_1, ..., t_n) = \beta(f)(\alpha_f(t_1), ..., \alpha_f(t_n))$.

Definition 3.2.8. Let D be a DRS, and L be the underlying first order language. Let $I = (A, \beta)$ be a partial L-interpretation. An assignment f verifies D if there is an assignment f' which extends f, and has the following property:

- f' is defined for all discourse referents of D, and for all variables which occur in relation or equivalence conditions of D.
- If $R(t_1, ..., t_n)$ is a DRS condition in D, then $\beta(R)(f'(t_1), ..., f'(t_n))$.
- If $t_1 \equiv t_2$ is a DRS condition in D, then $\alpha_{f'}(t_1) = \alpha_{f'}(t_2)$.
- If $\neg B$ is a DRS condition in D, then there is no assignment g which agrees with f' on all variables that are not discourse referents of B, such that g verifies B.
- If $B_1 \vee B_2$ is a DRS condition in D, then there is an assignment g which extends f' such that g verifies B_1 or g verfies B_2.
- If $B_1 \Rightarrow B_2$ is a DRS condition in D, then every assignment g which agrees with f' on all variables that are not discourse referents of B_1 and verifies B_1 also verifies B_2.

The PRS D is called true under I if the empty assignment verifies D.

3.2.3 Proof Representation Structures

PRSs are a special type of DRS. The difference lies in the definition of the *context* of a sentence. For a PRS, we consider the context of a sentence to be the accessible discourse referents, the mathematical formulas we are talking about, references to other parts of the discourse, and the premises which are active.

We will give the basic definitions for PRSs, based upon KOLEV's [13] Magister thesis. The definition of PRSs and PRS conditions is done via simultaneous recursion:

Definition 3.2.9. *Proof Representation Structures*
Let L be a first order language. A PRS is a quintuple $\langle id, D, M, C, R\rangle$, where

- id is a unique string.
- D is a set of discourse referents.
- M is a set of mathematical referents.
- C is a finite sequence of PRS conditions.
- R is a set of textual references.

and every element of M is either a term or a formula of L.

Definition 3.2.10. *PRS condition*
Let A, B be PRSs. Then

- $holds(X)$ is a PRS condition, where $X \in D$.
- $math\_id(X, Y)$ is a PRS condition, where $X \in D$ is a discourse referent and $Y \in M$ is a mathematical referent.
- A is a PRS condition.
- $\neg A$ is a PRS condition, representing a negation.
- $A := B$ is a PRS condition, representing a definition.
- $A \rightarrow B$ is a PRS condition, representing an implication or quantification.
- $A => B$ is a PRS condition, representing an assumption.
- *contradiction* is a PRS condition, representing a contradiction.

PRS conditions of the form $A := B$ are called *definition* conditions, conditions of the form $A \rightarrow B$ are called *implication* or *quantification* conditions, a condition of the form $A => B$ is also called a *assumption* condition, and a condition of the form *contradiction* is called a *contradiction* condition.

Remark. The first order language for PRSs which we use in the Naproche system is defined in the math lexicon which can be found in appendix B.

Definition 3.2.11. *Structure and non-structure PRS*
We distinguish between structure and non-structure PRS. A structure PRS is created when the word *"Theorem"* or *"Lemma"* is parsed in the input text. The id of a structure PRS starts with theorem/lemma and ends with a unique number. A structure PRS has exactly two conditions. Both of them are PRSs. The first PRS is called the goal, the second the body of the structure PRS. The idea is that the statement of, for example, a theorem is stored in the goal condition, while the proof of the theorem is stored in the body.

Every PRS which does not stem from the word *"Theorem"* or *"Lemma"*, and therefore does not have a theorem/lemma id, is called a non-structure PRS. If a structure PRS stems from the word *"Lemma"*, it is also called a *lemma* PRS. Analogously, if it stems from the word *"Theorem"*, then we call it a *theorem* PRS.

Definition 3.2.12. *Empty PRS*
Every PRS with no conditions is called *empty*. Analogously, every PRS with at least one conditions is called *non-empty*.

Definition 3.2.13. *The depth d of a PRS*
Again, we use simultaneous recursion. Let X be a discourse referent, Y be a mathematical referent, B,C be PRSs, and let B have conditions $\beta_1, ..., \beta_m$. We define the depth of a PRS recursively. Every empty PRS E has depth $d(E) = 0$. A non-empty PRS A with conditions $\gamma_1, .., \gamma_n$ has depth $d(A) = max_{i=1,...,n}\ d(\gamma_i) + 1$. The depth of a condition is defined as follows:

- $d(holds(X)) = 0$
- $d(math\_id(X, Y)) = 0$
- $d(B) = max_{j=1,...,m}\ d(\beta_j) + 1$ if B is non-empty.
- $d(B) = 0$ if B is empty.
- $d(\neg B) = d(B)$
- $d(B := C) = max(d(B), d(C))$
- $d(B \rightarrow C) = max(d(B), d(C))$
- $d(B => C) = max(d(B), d(C))$
- $d(contradiction) = 0$

Definition 3.2.14. *Accessibility*
A PRS B is accessibly from a PRS C if, and only if $B = C$ or B *subordinates* C.

Definition 3.2.15. *Subordinates*
A PRS B subordinates a PRS C if, and only if B immediately subordinates C, or if there is some PRS D such that B immediately subordinates D and D subordinates C.

Definition 3.2.16. *Immediately subordinates*
Let A be a PRS. A PRS B immediately subordinates a PRS C if, and only if

- B contains a condition of the form C.
- B contains a condition of the form $\neg C$.
- A has conditions $\gamma_1, ..\gamma_n$, and $B = \gamma_i$ and $C = \gamma_j$ for some $1 \leq i < j \leq n$.
- B contains a condition of the form $C := D$ or $D := C$ for some PRS D.
- B contains a condition of the form $C \rightarrow D$ or $D \rightarrow C$ for some PRS D.
- B contains a condition of the form $C \Rightarrow D$ or $D \Rightarrow C$ for some PRS D.
- $B := C$ is a condition in A.
- $B \rightarrow C$ is a condition in A.
- $B \Rightarrow C$ is a condition in A.

Definition 3.2.17. *The graphical representation of a PRS*
When we see PRSs only as quintuples, working with them becomes extremely tedious. In order to make it simpler, we developed a graphical notation for PRS, similar to the graphical notation for DRS.

The graphical notation of a PRS $P = \langle id, D, M, C, R\rangle$ with id i, discourse referents $d_1, ..., d_n$, mathematical referents $m_1, ..m_k$, conditions $c_1, ..., c_l$ and textual referents $r_1, ..r_p$ can be seen in figure 3.2.

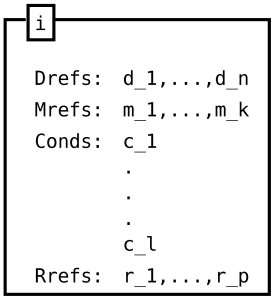

Figure 3.2: The graphical notation of a PRS $P = \langle id, D, M, C, R\rangle$ with id i, discourse referents $d_1, ..., d_n$, mathematical referents $m_1, ..m_k$, conditions $c_1, ..., c_l$ and textual referents $r_1, ..r_p$

Note that in the graphical notation we use $\rightarrow$ for assumption conditions, and $==>$ for implication conditions

The Semantic of PRS

In section 4.2 we define the *formula image* of a PRS, which maps a PRS to a set of sequences of first order formulas. This map gives an implicit definition of the PRS semantic.

The explicit definition is beyond the scope of this thesis, and will be covered in an separate article.

3.2.4 From Discourse to PRS

Having defined the Naproche language and PRSs, we will now take a look at the relation between them. We will show by example how the Naproche system transforms a text into a PRS. The explicit algorithm can be found in [13].

Our example is a short theorem from basic set theory which states that the empty set is an ordinal:

> Assume that $\neg\exists y\ y \in \emptyset$. Assume that for all x, $\neg x \in x$. Define $\mathit{Trans}(x)$ iff $\forall u \forall v((u \in v) \wedge (v \in x)) \rightarrow (u \in x)$. Define $\mathit{Ord}(x)$ iff $\mathit{Trans}(x) \wedge (\forall y((y \in x) \rightarrow \mathit{Trans}(y)))$.
>
> Theorem. $\mathit{Ord}(\emptyset)$.
> Proof. Consider $u \in v$ and $v \in \emptyset$. Assume that $\neg\mathit{Trans}(\emptyset)$. Then $\exists x(x \in \emptyset)$. Contradiction. Thus $\mathit{Trans}(\emptyset)$. Assume that $\neg\mathit{Trans}(v)$. Then $\exists x(x \in \emptyset)$. Contradiction. Thus $\mathit{Trans}(v)$. Thus $\mathit{Ord}(\emptyset)$. Qed.

This text is completely written in the Naproche language and can therefore be parsed by the Naproche system. In the following, we abbreviate discourse referent as Dref, and mathematical referent as Mref.

Like most DRS construction algorithms, and arguably human understanding, the PRS building process is incremental. We process one sentence after the other, interpreting and updating the context in each step. At the beginning of the parsing, there is no context, so we start with the empty PRS, as can be seen in figure 3.3.

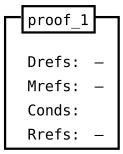

Figure 3.3: The $\mathit{Ord}(\emptyset)$ Example 1

The first sentence is an assumption:

> Assume that $\neg\exists y\ y \in \emptyset$.

A new *assumption* condition is added to the PRS. It has the form $A => B^*$, where A and B are also PRSs. The formula we assume to be true is written in the left-hand side PRS A of the *assumption* condition, and everything which is stated under this assumption is stored in the right-hand side PRS B of the *assumption* condition.

In our case, the formula is $\neg\exists y\ y \in \emptyset$. Internally, the Naproche system uses a different notation which will be introduced in 3.3. In this notation, the formula becomes $\sim (?[y] : (in(y, emptyset)))$. The Naproche system notices that there is a new entity in our discourse, namely this formula. Therefore, a new Dref is created. Since the entity is a formula we also get a new Mref. They are connected by a *math_id* condition. Finally, the Naproche system also adds a *holds* condition, to state that it considers this formula to be true. The corresponding PRS can be seen in figure 3.4.

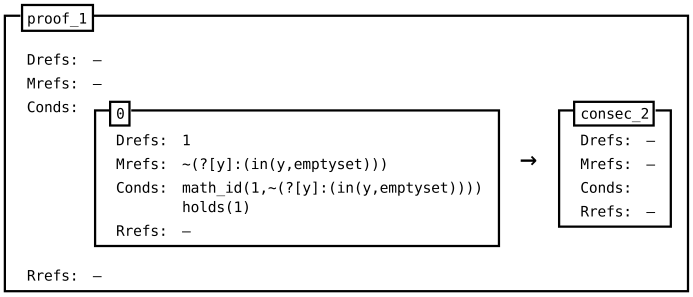

Figure 3.4: The $Ord(\emptyset)$ Example 2

Until we close the assumption by a *"Thus"* sentence, the contruction will proceed in the right-hand side PRS of the *assumption* condition. Next in our discourse is another assumption:

> Assume that for all x, $\neg x \in x$.

The Naproche system creates a new *assumption* condition. This time, our formula is *for all* x, $\neg x \in x$, which is not pure first order, but natural language mixed with first order. This formula gets stored as a *quantification* condition, which has the form $A \to B$ for PRSs A and B, in the left-hand side PRS of the *assumption* condition. The Naproche system creates a new *quantification* condition where x, the variable we quantify over, is stored in the the left-hand side PRS A of the *quantification* condition. The formula $\neg x \in x$ is saved in the right-hand side PRS B of the *quantification* condition. In the left-hand side PRS, x gets a discourse referent, and is saved as a Mref. Since x is only a variable, we do not have a *holds* condition. In the right-hand side PRS of the *quantification*

*Remember that in the graphical notation we use $\to$ for *assumption* conditions, and ==> for *implication* and *quantification* conditions

condition, we only have one new Dref, but two Mrefs. The Mrefs are our formula, $\neg x \in x$, and the free variable in the formula, x. Now, x is already known and accessible, and therefore does not get a new Dref. The formula is treated exactly as we saw in the last assumption. The corresponding PRS for our parsed discourse can be seen in figure 3.5.

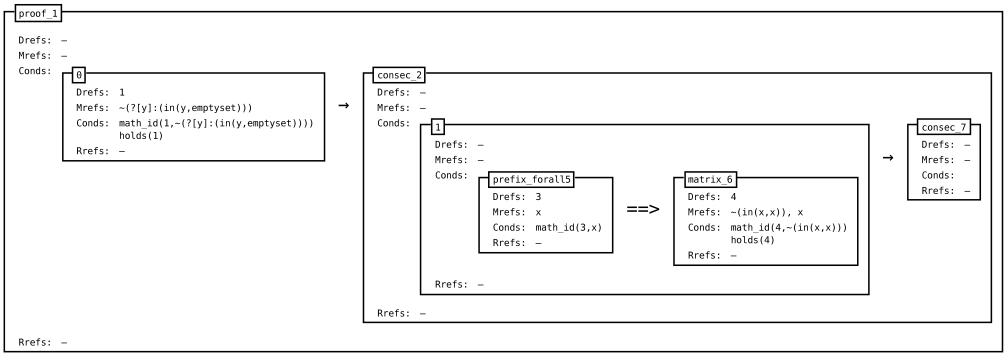

Figure 3.5: The $Ord(\emptyset)$ Example 3

Because we parsed another assumption, the construction will proceed in the right-hand side PRS of the newly added *assumption* condition. Next, we have two definitions:

Define $Trans(x)$ iff $\forall u \forall v((u \in v) \wedge (v \in x)) \rightarrow (u \in x)$.
Define $Ord(x)$ iff $Trans(x) \wedge (\forall y((y \in x) \rightarrow Trans(y)))$.

For each of them, a *definition* condition is created. We will consider the $Trans(x)$ definition in detail.

On the left-hand side PRS of the *definition* condition is the new entity which we want to define, $Trans(x)$. We have two new Drefs, and two Mrefs. The free variable x is treated as a new discourse referent because the x we encountered before it not accessibly. *math_id* conditions connect the Mrefs to the Drefs, and we have a *holds* condition for the formula.

The right-hand side PRS contains the formula $\forall u \forall v((u \in v) \wedge (v \in x)) \rightarrow (u \in x)$. Our Mrefs are the formula itself, and its free variable, x. x does not get a new discourse referent as we already have an accessibly x. Again, we have a *math_id* and a *holds* statement for the formula.

The $Ord(x)$ definition is treated analogously. Figure 3.6 shows the newly added *definition* conditions.

Note that we are still working within the right-hand side PRS of the second *assumption* condition.

The next sentence is simply "Theorem.". This is a structure marker in the Naproche language. For now, it just creates a new empty PRS as condition , a *theorem* PRS. The updated PRS can be seen in figure 3.7.

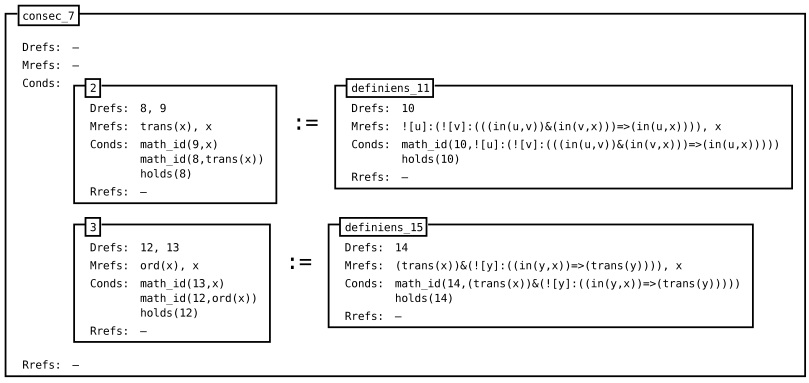

Figure 3.6: The $Ord(\emptyset)$ Example 4

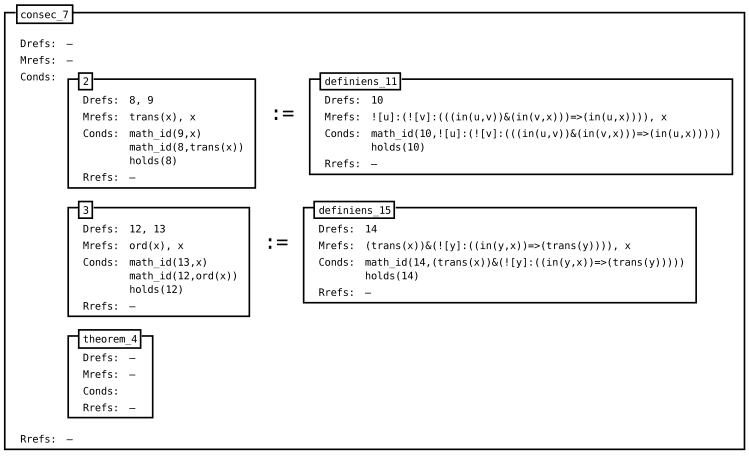

Figure 3.7: The $Ord(\emptyset)$ Example 5

The PRS construction proceeds within the *theorem* PRS.

The statement of our theorem is simply a formula, $Ord(\emptyset)$. The Naproche system creates a new PRS as condition in the *theorem* PRS. In this new PRS, the statement, or goal, of the theorem is stored, and therefore the PRS is called the *goal* condition of the *theorem* PRS. (Figure 3.8).

The sentence "Proof." lets the Naproche system close the *goal* condition, and start the second PRS in our *theorem* PRS which is called the *body* condition. The *body* condition will store the proof of the theorem. Figure 3.9 shows the updated PRS.

We now consider the first part of the proof:

> Consider $u \in v$ and $v \in \emptyset$. Assume that $\neg\, Trans(\emptyset)$. Then $\exists x(x \in \emptyset)$.
> Contradiction. Thus $Trans(\emptyset)$.

We start with an assumption, "Consider $u \in v$ and $v \in \emptyset$". The corresponding PRS has four new Drefs, two for the formulas , and two for the free variables u

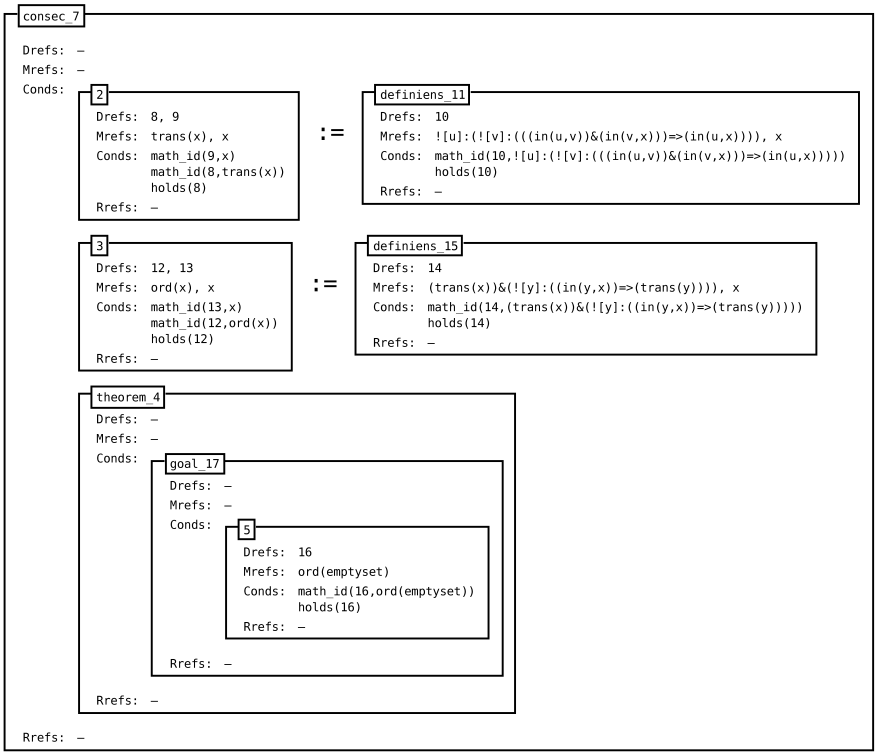

Figure 3.8: The $Ord(\emptyset)$ Example 6

and v. Note that $\emptyset$ does not get a Dref because it is a constant.

The second assumption brings nothing new. The "Then ..." sentence creates a new condition which is a PRS. In this PRS, the formula after the "Then" is stored. The "Contradiction." sentence becomes a PRS with a *contradiction* condition.

The final sentence of this part, "Thus $Trans(\emptyset)$." closes the last assumption, namely "Assume that $\neg Trans(\emptyset)$.". The formula after "Thus" is treated as usual.

The body of the *theorem* PRS can be seen in figure 3.10.

We only have a few sentences left to parse:

> Assume that $\neg Trans(v)$. Then $\exists x(x \in \emptyset)$. Contradiction. Thus $Trans(v)$. Thus $Ord(\emptyset)$. Qed.

The remainder of the proof brings only one novelty, the "Qed." sentence. "Qed." is a marker which lets the Naproche system close all assumptions which were opened in the *body* condition, the *body* condition itself, and also the *theorem* PRS. If there were any more sentences, then the construction would proceed in the PRS which has the *theorem* PRS as condition. So, the sentence "Qed." is similar to a "Thus" sentence. "Thus" closes the last assumption, and "Qed." closes a *theorem* PRS. Figure 3.11 shows the complete *theorem* PRS.

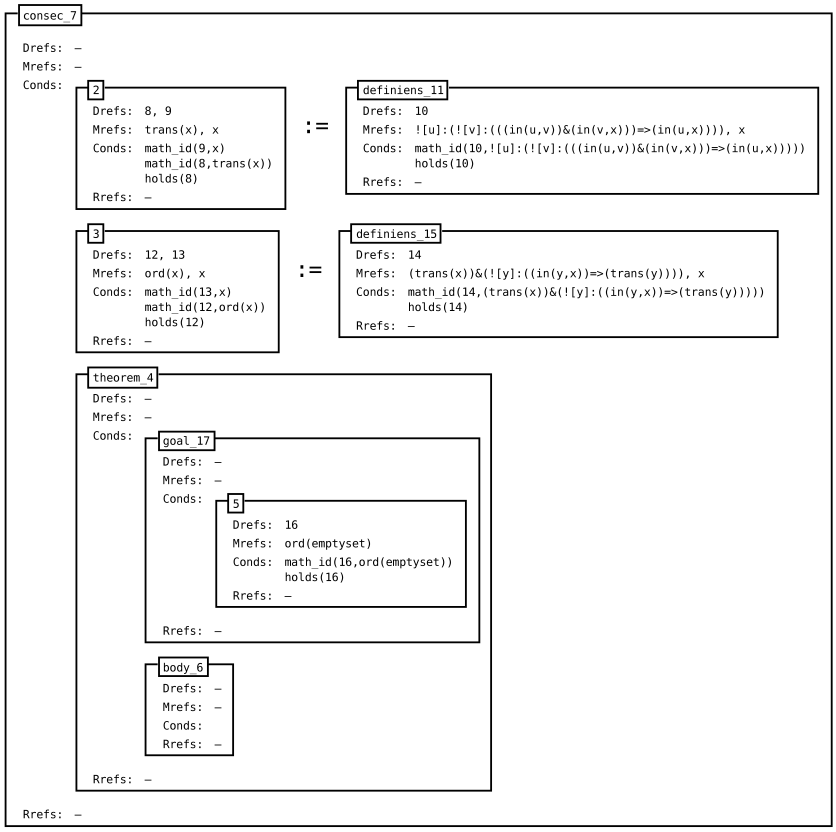

Figure 3.9: The $Ord(\emptyset)$ Example 7

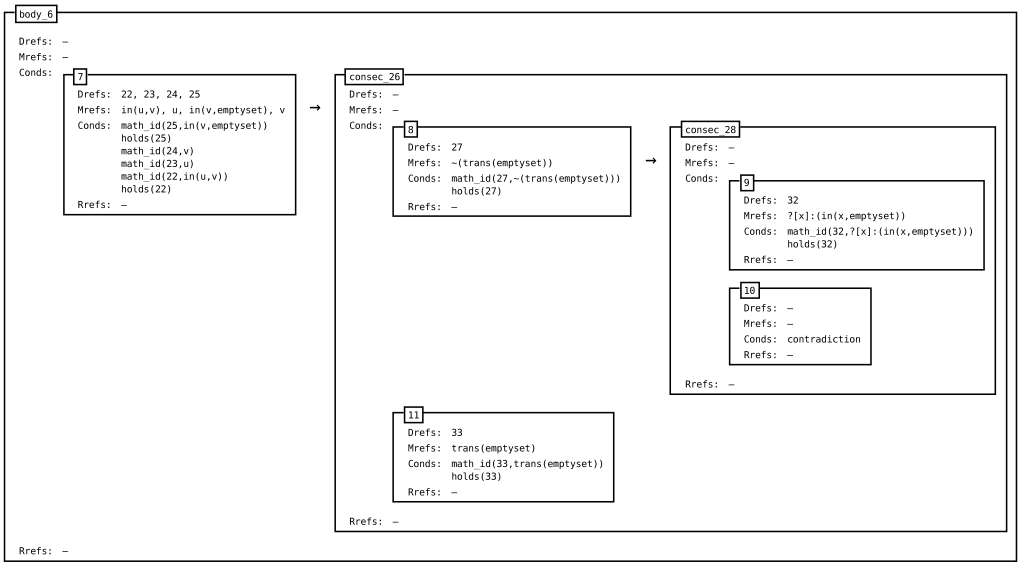

Figure 3.10: The $Ord(\emptyset)$ Example 8

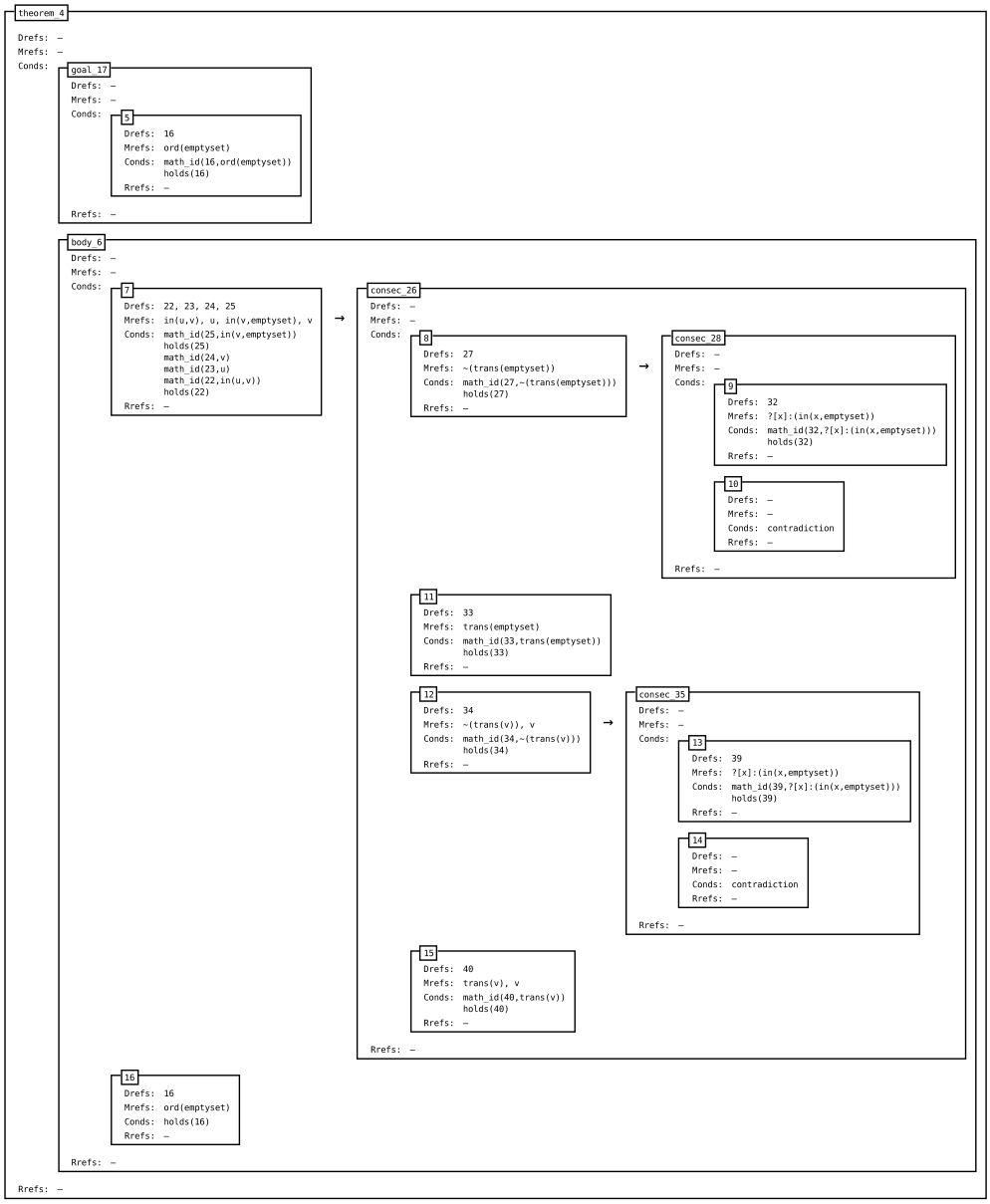

Figure 3.11: The $Ord(\emptyset)$ Example 9

Finally, figure 3.12 shows the complete PRS for the proof of $Ord(\emptyset)$.

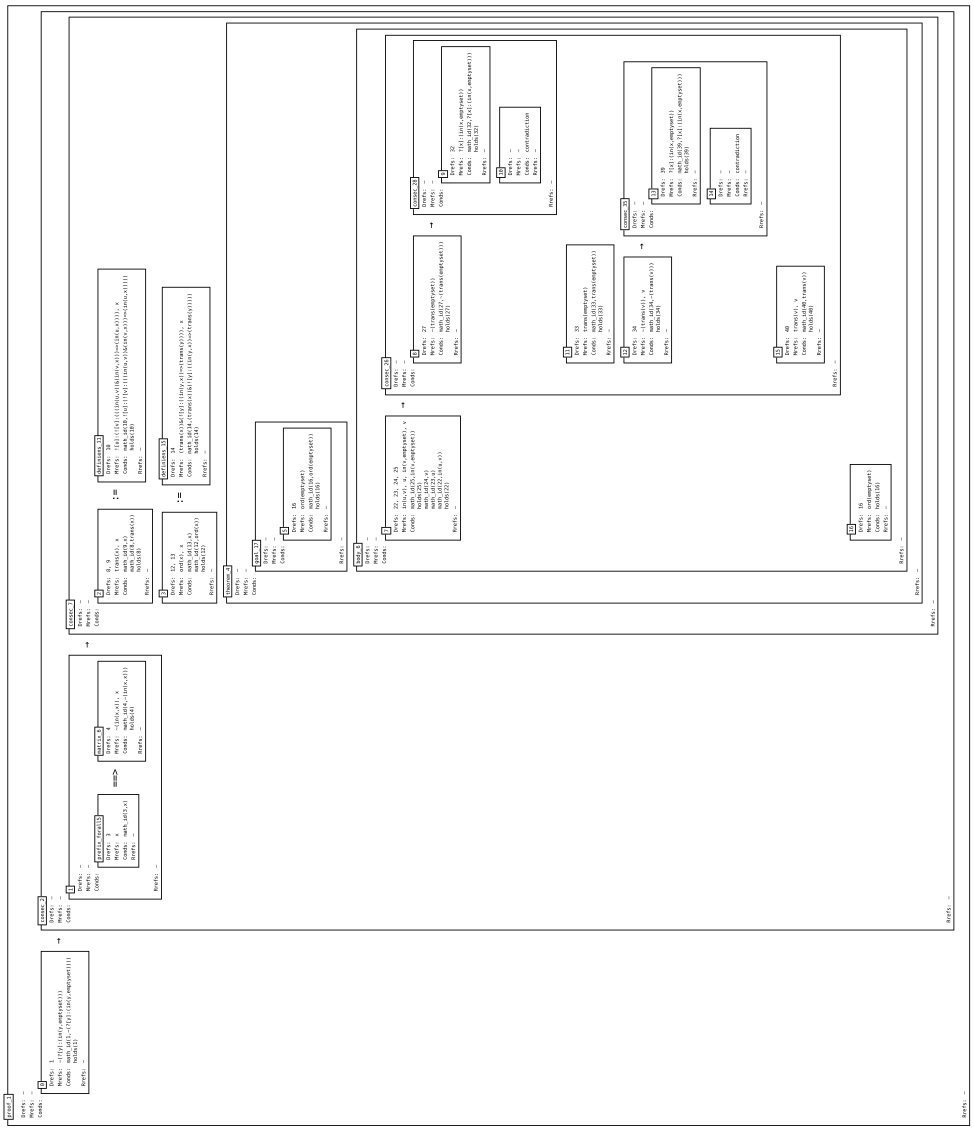

Figure 3.12: The PRS for the proof of $Ord(\emptyset)$.

3.3 Automated Theorem Provers and TPTP

The idea that computers could be used to check, or even generate, proofs goes back to the 1960s. John McCarthy's paper from 1963 [17] talks about *"proof procedures and proof checking procedures"*:

> *... computers can be used to carry out the algorithms that are being devised to generate proofs of sentences in various formal systems.* [17]

The Mizar project was started in 1973, and even the idea for the SAD program goes back to around 1970 [22].

By the end of the 20th century, several Automated Theorem Provers existed. Naturally, scientists were interested in comparing them. In order to do that, one first needs test cases. In [27], Geoff Sutcliffe brought up the idea of the TPTP (Thousands of Problems for Theorem Provers) Problem Library.

> *The TPTP (Thousands of Problems for Theorem Provers) Problem Library is a library of test problems for automated theorem proving (ATP) systems.* [23]

and further,

> *The principal motivation for the TPTP is to support the testing and evaluation of ATP systems, to help ensure that performance results accurately reflect the capabilities of the ATP system being considered. A common library of problems is necessary for meaningful system evaluations, meaningful system comparisons, repeatability of testing, and the production of statistically significant results. The TPTP is such a library.* [23]

In 1996 [28], the first ATP system competition was devised, and it was held in 1997 [26]. The TPTP library comes with its own syntax for writing down problems. The current version can be found on the TPTP homepage [29]. Over the years, several programs, which deal with TPTP in one form or another, have been developed. One program, which is of particular interest for the Naproche system, is SystemsOnTPTP [24]. It translates problems, written in TPTP Syntax, into the input format of the desired ATP, for example OTTER or Vampire. This means, that if you have a problem in TPTP Syntax, you can run it on several different provers, using this program.

TPTP and Naproche

In the Naproche system, we want to check mathematical texts with ATPs. The TPTP Syntax, in combination with SystemsOnTPTP [24] makes us independent

from specific provers. We can use different provers without having to worry about different input formats. Furthermore, it allows us to compare different provers easily and see which one is best suited for our efforts. Once we start writing in, or translating into, the Naproche language, this would also be a source of new problems for the TPTP library.

3.3.1 The TPTP Syntax

We will give a quick overview of the part of the TPTP Syntax which is used in the Naproche system. If you are interested in the original definition, take a look at [29].

A first order formula in TPTP has the following form:

$$fof(\langle name\rangle, \langle formula\_role\rangle, \langle fof\_formula\rangle\langle annotations\rangle).$$

Here, $\langle name\rangle$ is a string. Usually, we use a number for $\langle name\rangle$. The combination $(\langle name\rangle, \langle formula\_role\rangle)$ must be unique.

$\langle formula\_role\rangle$ defines the role of the formula. In the Naproche system, we only allow two values here: *axiom* for formulas which we do not want to prove, and *conjecture* for formulas we want to prove.

$\langle fof\_formula\rangle$ is the actual first order formula, written in the TPTP syntax which will be defined later in this section.

$\langle annotations\rangle$ is used for additional information. In the Naproche system, we do not use this field.

Before we go on, and explain $\langle fof\_formula\rangle$, let's look at an actual Naproche TPTP obligation:

Example 3.3.1. Naproche TPTP Query

```
fof(1, axiom, ~(?[Vy]:(in(Vy,vemptyset)))).
fof(2, axiom, ![Vx]:(~(in(Vx,Vx)))).
fof(3, axiom, ![Vx]:((trans(Vx))<=>(![Vu]:(![Vv]:
  (((in(Vu,Vv))&(in(Vv,Vx)))=>(in(Vu,Vx)))))).
fof(4, axiom, ![Vx]:((ord(Vx))<=>((trans(Vx))&(![Vy]:
  ((in(Vy,Vx))=>(trans(Vy))))))).
fof(5, axiom, in(vv,vemptyset)).
fof(6, axiom, in(vu,vv)).
fof(7, axiom, ~(trans(vemptyset))).
fof(1, conjecture, ?[Vx]:(in(Vx,vemptyset))).
```

The Naproche system asks the prover whether it can prove conjecture 1 from the axiom 1-7.

Formulas in TPTP

The TPTP language for formulas has the usual logical connectives: $\sim$ for negation, & for conjunction, | for disjunction, => for implication and <=> for equivalence. Quantifiers are !, for universal quantification, and ?, for existential quantification.

Each string starting with an upper case letter is a variable. Strings starting with a lower case letter are either constants or relations. Relations are followed by brackets, which contain variables, constants or other relations. Round brackets have their usual meaning, square brackets are used with quantifiers to denote the variables which are bound by the quantifier. For an example, let us consider the second axiom from the example Naproche TPTP query:

$$fof(2, axiom, ![Vx] : (\sim (in(Vx, Vx)))).$$

The formula here is $![Vx] : (\sim (in(Vx, Vx)))$. Substituting the TPTP symbols with our normal symbols gives $\forall[Vx] : (\neg(in(Vx, Vx)))$. Now, in is the predicate we use for $\in$, and V is the letter the Naproche system attaches to bound variables. So, further simplification leads to $\forall x, \neg x \in x$, the foundation axiom.

Free and bound variables in TPTP and the Naproche system

In natural language mathematics, free and bound variables are usually not specifically marked or distinguished. In TPTP syntax, however, every variable must be bound! Consequently, we can have no free variables in a formula which is written in TPTP syntax. Further, variables must start with a capital letter in TPTP, whereas in natural language, they have no restrictions.

The Naproche system deals with this as follows. When it translates a Naproche formula into a TPTP formula it first uses the algorithm, described in 2.1.6, to calculate the free and bound variables. Then, a lower case v is attached to each free variable and to each constant, and an upper case V is attached to each bound variable. So, free variables in Naproche formulas are treated as constants in TPTP. As an example, consider the conjecture of the example query:

$$?[Vx] : (in(Vx, vemptyset))$$

In our first order logic, this formula reads: $\exists x, x \in \emptyset$. The bound variable x became Vx in the TPTP translation, and the constant *emptyset* became *vemptyset* to ensure that it is treated as a constant by the ATP.

3.4 An Example: The Burali-Forti Paradox

In 1897, Burali-Forti discovered that the class of all ordinals is not a set [2]. Together with Russell's antinomy [9], these two theorems are prominent examples of why in modern set theory one distinguishes classes and sets. Difficulties

like these lead mathematicians to abandon FREGE's set theory, and to develop and use ZFC.

We can formulate the BURALI-FORTI paradox in the Naproche language by using VON NEUMANN's definition of ordinals in the proof. Once it is stated in the Naproche language, the Naproche system can successfully check it. We present the BURALI-FORTI paradox in the Naproche language:

Assume that $\neg\exists y\ (y \in \emptyset)$.
Assume that $\forall x\ (\neg x \in x)$.
Define $\mathit{Trans}(x)$ if, and only if $\forall u, v((u \in v) \wedge (v \in x)) \rightarrow (u \in x))$.
Define $\mathit{Ord}(x)$ if, and only if $\mathit{Trans}(x) \wedge (\forall y(y \in x) \rightarrow \mathit{Trans}(y))$.

Theorem.
$\mathit{Ord}(\emptyset)$.

Proof.
Consider $u \in v$ and $v \in \emptyset$.
Assume that $\neg\, \mathit{Trans}(\emptyset)$.
Then $\exists x(x \in \emptyset)$.
Contradiction.
Thus $\mathit{Trans}(\emptyset)$.
Assume that $\neg\, \mathit{Trans}(v)$.
Then $\exists x(x \in \emptyset)$.
Contradiction.
Thus $\mathit{Trans}(v)$.
Thus $\mathit{Ord}(\emptyset)$.
Qed.

Theorem.
For all x, y $x \in y \wedge \mathit{Ord}(y)$ implies $\mathit{Ord}(x)$.

Proof.
Consider $x \in y$ and $\mathit{Ord}(y)$.
Then $\forall x((x \in y) \rightarrow \mathit{Trans}(x))$.
Hence $\mathit{Trans}(x)$.
Assume that $u \in x$.
Then $u \in y$.
Hence $\mathit{Trans}(u)$.
Thus $\mathit{Ord}(x)$.
Qed.

Theorem.

For all $x, \neg(\forall u(u \in x) \leftrightarrow \mathit{Ord}(u))$.

Proof.
Assume for a contradiction that there is an x such that $\forall u((u \in x) \leftrightarrow \mathit{Ord}(u))$.

Lemma.
$\mathit{Ord}(x)$
Proof.
Let $u \in v$ and $v \in x$.
Then $\mathit{Ord}(v)$.
Hence $\mathit{Ord}(u)$.
So, $u \in x$.
Thus $\mathit{Trans}(x)$.
Let $v \in x$.
Then $\mathit{Ord}(v)$.
So $\mathit{Trans}(v)$.
Thus $\mathit{Ord}(x)$.
Qed.

Then $x \in x$.
Contradiction.
Qed.

A Naproche text needs to be self-containing which means that everything which is used in the text must be defined in it. Therefore, we need the assumptions and definitions. The two theorems which are proven before Burali-Forti's paradox are stated to make the task easier for the ATP. We could skip them, and only write down the proof of the main theorem, but this would either increase the time needed to check the text, or even render the Naproche system completely unable to do so.

If you like to have more details on how Naproche checks this text, take a look at [14]. This example is covered there in great detail.

Chapter 4

The Naproche Calculus

In this chapter, we define our calculus for Proof Representation Structures. We take a look at the relation between the Naproche language and the calculus, as well as the relation between the calculus and the actual sourcecode of the Naproche system. In the last section, we show its soundness and completeness under certain conditions.

Note that the notion of a calculus might be a bit misleading. The calculus we are going to introduce is not purely on PRS level, like for example the DRS calculus in [30], but rather a mixture of PRS and first order arguments. We decided to use the word calculus nonetheless, because the Naproche calculus provides a set of rules which allow to create certain objects, PRSs, and such derived objects are 'correct' in a certain sense. In this aspect, the Naproche calculus is similar to other calculi.

4.1 Preliminaries

Before we define the calculus, we need some preliminary definitions and lemmas. Note that easy formal proofs in the sequent calculus are not annotated in this chapter.

Definition 4.1.1. Let A,B be sets. Then $A - B$ is defined as

$$A - B = \{x \mid x \in A \wedge x \notin B\}$$

Definition 4.1.2. For a sequent $\langle \varphi_1, ..., \varphi_n \rangle$ we define

$$\bigwedge \langle \varphi_1, ..., \varphi_n \rangle = \varphi_1 \wedge ... \wedge \varphi_n$$

and

$$\neg \langle \varphi_1, ..., \varphi_n \rangle = \langle \neg \varphi_1, ..., \neg \varphi_n \rangle$$

In the special case of the empty sequence, we set $\bigwedge\langle\rangle = \top$ and $\neg\langle\rangle = \langle\rangle$. We also use the more common notation for the $\bigwedge$ operator: Let $\varphi_1, ..., \varphi_n$ be formulas. We define:

$$\bigwedge_{i=1}^{n} \varphi_i = \varphi_1 \wedge ... \wedge \varphi_n$$

For two sequents $\langle\varphi_1, ..., \varphi_n\rangle$ and $\langle\psi_1, ..., \psi_m\rangle$ we define

$$\langle\varphi_1, ..., \varphi_n\rangle + \langle\psi_1, ..., \psi_n\rangle = \langle\varphi_1, ..., \varphi_n, \psi_1, .., \psi_n\rangle$$

Let $\Gamma_1, .., \Gamma_n$ be sequents. Then we define $\bigoplus_1^0 \Gamma_i = \langle\rangle$ and

$$\bigoplus_{i=1}^{n} \Gamma_i = \bigoplus_{i=1}^{n-1} \Gamma_i + \Gamma_n$$

Definition 4.1.3. Let Γ and $\Theta = \langle\theta_1, ..., \theta_n\rangle$ be sequents. We define $\Gamma \vdash_{seq} \Theta$ if, and only if for all $1 \leq i \leq n$ holds

$$\Gamma + \langle\theta_j \mid 1 \leq j \leq i-1\rangle \vdash \theta_i$$

Lemma 4.1.1. *Let $\Gamma \vdash_{seq} \langle\theta_1, ..., \theta_n\rangle$, then $\Gamma \vdash \theta_i$ for all $1 \leq i \leq n$.*

Proof: By induction over the length n of the sequence. For $n = 1$ the lemma is obvious. So let $n > 1$ and assume the result is true for all $m < n$. By definition, $\Gamma \vdash_{seq} \langle\theta_1, ..., \theta_n\rangle$ implies $\Gamma \vdash_{seq} \langle\theta_1, ..., \theta_{n-1}\rangle$. Using the induction hypothesis, we get $\Gamma \vdash \theta_i$ for all $1 \leq i \leq n-1$. It remains to show, that $\Gamma \vdash \theta_n$. We use the definition of $\vdash_{seq}$ and calculate:

$$\dfrac{\dfrac{\Gamma + \langle\theta_i \mid 1 \leq i \leq n-2\rangle \quad \theta_{n-1} \qquad \Gamma + \langle\theta_i \mid 1 \leq i \leq n-2\rangle \quad \theta_{n-1} \qquad \theta_n}{\Gamma + \langle\theta_i \mid 1 \leq i \leq n-2\rangle \quad \theta_{n-1} \rightarrow \theta_n}}{\Gamma + \langle\theta_i \mid 1 \leq i \leq n-2\rangle \quad \theta_n}$$

So, we shortened the sequence which we had to add to Γ by one: We concluded $\Gamma + \langle\theta_i \mid 1 \leq i \leq n-2\rangle \vdash \theta_n$ from $\Gamma + \langle\theta_i \mid 1 \leq i \leq n-1\rangle \vdash \theta_n$. Iterated application of this argument gives us $\Gamma \vdash \theta_n$. *qed*

Lemma 4.1.2. *Let $\Gamma \vdash \theta_i$ for all $1 \leq i \leq n$, then $\Gamma \vdash_{seq} \langle\theta_1, ..., \theta_n\rangle$.*

Proof: Follows directly from the monotonicity rule. *qed*

Lemma 4.1.3. *Let* Γ, Θ *and* $\Phi = \langle \varphi_1, .., \varphi_n \rangle$ *be sequents and* $\Gamma + \Theta \vdash_{seq} \Phi$. *Then*

$$\Gamma \vdash_{seq} \langle \bigwedge \Theta \to \varphi_i \mid 1 \leq i \leq n \rangle$$

Proof: First note that by lemma 4.1.1 we get that $\Gamma + \Theta \vdash \varphi_i$ for all $1 \leq i \leq n$. By lemma 4.1.2, it is enough to show, that $\Gamma \vdash \bigwedge \Theta \to \varphi_i$, for all $1 \leq i \leq n$. Let $\Theta = \langle \theta_1, ..., \theta_m \rangle$. We show that $\Gamma + \Theta \vdash \varphi_i$ implies $\Gamma \vdash \bigwedge \Theta \to \varphi_i$, by induction on m. For $m = 1$, the statement is obvious:

Γ	θ_1	φ_i
Γ	$\theta_1 \to \varphi_i$	

Now, let $m > 1$ and assume that $\Gamma + \langle \theta_1, ..., \theta_l \rangle \vdash \varphi_i$ implies $\Gamma \vdash \bigwedge_{j=1}^{l} \theta_j \to \varphi_i$ for all $l < m$. Let $\Gamma + \langle \theta_1, ..., \theta_m \rangle \vdash \varphi_i$. Using the induction hypothesis, we get $\Gamma + \langle \theta_1 \rangle \vdash \bigwedge_{j=2}^{m} \theta_j \to \varphi_i$. Setting $\psi = \bigwedge_{j=2}^{m} \theta_j$, $\theta = \theta_1$, and $\varphi = \varphi_i$ we can calculate:

Γ	θ	$\psi \to \varphi$		(1)
Γ	$\theta \to (\psi \to \varphi)$		$\to$ Intro.	(2)
Γ	$\neg(\theta \to \neg\psi)$	$\neg(\theta \to \neg\psi)$	Assump.	(3)
Γ	$\neg(\theta \to \neg\psi)$	θ	(3) + Lemma 2.3.4	(4)
Γ	$\neg(\theta \to \neg\psi)$	$\theta \to (\psi \to \varphi)$	(2) + Mono.	(5)
Γ	$\neg(\theta \to \neg\psi)$	$\psi \to \varphi$	(4,5) + $\to$ Ele.	(6)
Γ	$\neg(\theta \to \neg\psi)$	ψ	(3) + Lemma 2.3.5	(7)
Γ	$\neg(\theta \to \neg\psi)$	φ	(6,7) + $\to$ Ele.	(8)
Γ	$\neg(\theta \to \neg\psi) \to \varphi$		(8) + $\to$ Intro.	(9)

Since $\neg(\theta \to \neg\psi) = \theta \wedge \psi \Leftrightarrow \bigwedge_{j=1}^{m} \theta_j$, we can conclude that $\Gamma \vdash \bigwedge_{j=1}^{m} \theta_j \to \varphi_i$.

qed

Lemma 4.1.4. *Let* Γ, Θ, Φ *be sequents and* $\Gamma \vdash_{seq} \Theta$, $\Gamma + \Theta \vdash_{seq} \Phi$. *Then* $\Gamma \vdash_{seq} \Phi$.

Proof: Let $\Theta = \langle \theta_i \mid 1 \leq i \leq n \rangle$ and $\Phi = \langle \varphi_j \mid 1 \leq j \leq m \rangle$. By Lemma 4.1.1, we get that $\Gamma \vdash \theta_i$ for all $1 \leq i \leq n$. We show that $\Gamma \vdash \varphi_j$ for all $1 \leq j \leq m$. Fix a j and assume that $\Gamma \vdash \varphi_l$ for all $l < j$. We calculate:

Γ	$\theta_1, ...\theta_n, \varphi_1, ...\varphi_{j-1}$	φ_j
Γ	$\theta_1, ...\theta_n, \varphi_1, ...\varphi_{j-2}$	$\varphi_{j-1} \rightarrow \varphi_j$
$\vdots$	$\vdots$	$\vdots$
Γ	$\theta_1 \rightarrow (...(\varphi_{j-1} \rightarrow \varphi_j)...)$	

Iterated application of the $\rightarrow$ eliminations rule and the induction hypothesis gives $\Gamma \vdash \varphi_j$. Hence, $\Gamma \vdash \varphi_j$ for all $1 \leq j \leq m$. But by lemma 4.1.2, this implies $\Gamma \vdash_{seq} \Phi$. *qed*

4.2 A Calculus for PRSs

In order to define our calculus for PRSs, we first need to introduce the notion of the *Formula Image* of a PRS. Remember that a PRS only has finitely many conditions (3.2.10).

For this chapter, we assume that in every PRS, for each Dref which is introduced, there is exactly one *math_id* condition. The *math_id* condition for a Dref is in the same PRS in which that Dref is introduced. If a Dref is not introduced in a PRS, then there is no *math_id* condition for this Dref in that PRS. This implies, that if there is a *holds* condition with argument x in a PRS A then there is PRS B which is accessible from A, and B has a *math_id* condition which has x as the Dref argument. So, each Dref is mapped to a term or a formula by a *math_id* condition. Because there is exactly one *math_id* condition for each Dref, this map is well-defined. Using this map, we simplify our notation, and assume that *holds* conditions do not have Drefs, but formulas as arguments. Note that, the PRS construction algorithm which is used in the current version of the Naproche system ensures that there is exactly one *math_id* condition for each Dref.

Definition 4.2.1. *Mref*(A)
Let A be a PRS. Then $Mref(A)$ is the set of all Mrefs of A.

As an example, consider the sentence *"For all x x=x"*. The corresponding PRS has one *quantification* condition of the form $B \rightarrow C$ for some PRSs B, C. B has only one condition: $math\_id(n, x)$, for some integer n. C is a PRS with $math\_id(m, x = x)$ and $holds(m)$ as conditions, for an integer m. In this example, $Mref(B) = \{x\}$, and $Mref(C) = \{x, x = x\}$. Note, that the PRS contruction ensures that there is at least one variable between *"For all"* and the formula.

Definition 4.2.2. *Quantifier over a set*
Let $X = \{x_1, ..., x_n\}$ be a finite set. Then we define $\forall X\ \varphi$ as $\forall x_1, ..., x_n\ \varphi$. If all x_i are variables, and φ is a formula, then $\forall X\ \varphi$ is also a formula.

Note that we assumed that we have a way of ordering the elements of the set X, which is normally not the case. This could cause a problem because for

example, if $X = \{x, y\}$, then $\forall X\ \varphi$ could be either $\forall x, y\ \varphi$ or $\forall y, x\ \varphi$. But since the order of the variables in simultaneous quantification is not important, all possible interpretations are equivalent.

For example, let A be a PRS with $Mref(A) = \{x_1, ..., x_n\}$. Then $\forall\ Mref(A)\ \varphi$ stands for $\forall x_1, ..., x_n\ \varphi$. We will use this, when we define the *Formula Image* of a PRS with *quantification* conditions.

Definition 4.2.3. *The Formula Image FI of a PRS*
Let E be an empty PRS, A be a non-empty, non-structure PRS with conditions $\gamma_1, .., \gamma_n$. Let B,C be non-structure PRSs, T be a theorem PRS with goal G_T and body B_T, and L be a lemma PRS with goal G_L and body B_L.

FI is a map from the set of PRSs into the set of sequences of first order formulas.

- $FI(E) = \langle \top \rangle$
- $FI(A) = \bigoplus_{i=1}^{n} \beta(\gamma_i)$
- $FI(T) = FI(G_T)$
- $FI(L) = FI(G_L)$.

Where β maps PRS conditions to sequences of first order formulas as follows:

- $\beta(holds(\varphi)) = \langle \varphi \rangle$
- $\beta(math\_id(X, Y)) = \langle\rangle$ for all Drefs X, and all Mrefs Y.
- $\beta(B) = FI(B)$
- $\beta(\neg B) = \neg FI(B)$
- $\beta(B := C) = \langle \forall\ free(\bigwedge FI(B) \leftrightarrow \bigwedge FI(C)) - X\ \bigwedge FI(B) \leftrightarrow \bigwedge FI(C) \rangle$. X is the set of all free variables of the premises of this definition condition.
- $\beta(B \rightarrow C) = \langle \forall\ Mref(B) - X\ \varphi_i \mid \langle \varphi_i \rangle = FI(C) \rangle$ if $FI(B) = \langle\rangle$. X is the set of all free variables of the premises of this implication condition.
- $\beta(B \rightarrow C) = \langle \forall\ free(\bigwedge FI(B)) - X\ \bigwedge FI(B) \rightarrow \varphi_i \mid \langle \varphi_i \rangle = FI(C) \rangle$ if $FI(B) \neq \langle\rangle$. X is the set of all free variables of the premises of this implication condition.
- $\beta(B => C) = \langle \forall\ free(\bigwedge FI(B)) - X\ \bigwedge FI(B) \rightarrow \varphi_i \mid \langle \varphi_i \rangle = FI(C) \rangle$, if C has conditions $\varphi_1, .., \varphi_n$ and $\varphi_n \neq contradiction$. X is the set of all free variables of the premises of this assumption condition.

- $\beta(B \Rightarrow C) = \langle \neg \bigwedge FI(B) \rangle$, if C has conditions $\varphi_1, .., \varphi_n$ and $\varphi_n = contradiction$
- $\beta(contradiction) = \bot$

Note that as stated in the introduction, we assume that *holds* conditions have formulas as arguments.

The definition of the formula image of a PRS is not complete without the definition of the premises of a PRS condition:

Definition 4.2.4. *The Premises of PRSs and PRS conditions*
Let Θ be a finite sequence of first order formulas. Let A be a PRS with conditions $\gamma_1, .., \gamma_n$. Let $1 \leq i \leq n$, and B be the PRS with conditions $\gamma_1, .., \gamma_{i-1}$.

The premises of a PRS, or a PRS condition, is a finite sequence of first order formulas. Unless noted otherwise, the premises of a PRS is the empty sequence. If the PRS A has the premises Θ then the premises of the PRS condition γ_i is the sequence $\Theta + FI(B)$. Furthermore, we define for a PRS condition γ with premises Θ:

- If $\gamma = B$ for a non-structure PRS B, the premises of B is Θ.
- If $\gamma = \neg B$ for a PRS B, the premises of B is Θ.
- If γ is a definition condition $B := C$, the premises of B and C is Θ.
- If γ is an implication condition $B \rightarrow C$, the premises of B is Θ, and the premises of C is $\Theta + FI(B)$.
- If γ is an assumption condition $B \Rightarrow C$, the premises of B is Θ, and the premises of C is $\Theta + FI(B)$.
- If γ is a theorem PRS T,with body B and goal G, the premises of B is Θ, and the premises of G is $\Theta + FI(B)$.
- If γ is a lemma PRS L, with body B and goal G, the premises of B is Θ, and the premises of G is $\Theta + FI(B)$.

The Calculus Definition

The Naproche calculus, which we are going to define now, is based on an already given first order calculus P. Note that in the Naproche system, this calculus is implemented by an ATP. Fix a first order calculus P for this section.

Definition 4.2.5. *Derivable in a Calculus*
Let Γ be a sequence of formulas and φ be a formula. We write $\Gamma \vdash^P \varphi$ if we can derive φ from the formulas in Γ in P. For another sequence of formulas $\Psi = \langle \psi_1, .. \psi_n \rangle$, we write $\Gamma \vdash^P_{seq} \Psi$ if $\Gamma \vdash^P \psi_i$ is true for all $1 \leq i \leq n$.

Definition 4.2.6. *The Naproche Calculus*
Let Θ be a finite sequence of formulas. As described at the beginning of this section, we assume that *holds* conditions have a formula as argument.

We define derivability under premises for PRSs: Let A be a PRS with conditions $\gamma_1, ..., \gamma_n$, which is derivable under the premises Θ.

- *Empty PRS*
  Any empty PRS is derivable under all premises.

- *holds condition*
  Let φ be a formula such that $\Theta + FI(A) \vdash^P \varphi$.
  Then the PRS B with conditions $\gamma_1, ..., \gamma_n, \mathit{holds}(\varphi)$ is derivable under the premises Θ

- *non-structure PRS condition*
  Let C be a PRS, which is derivable under the premises $\Theta + FI(A)$.
  Then the PRS B with conditions $\gamma_1, ..., \gamma_n, C$ is derivable under the premises Θ.

- *$\neg$PRS condition*
  Let C be a PRS and $\Theta + FI(A) \vdash^P_{seq} \neg FI(C)$.
  Then the PRS B with conditions $\gamma_1, ..., \gamma_n, \neg C$ is derivable under the premises Θ.

- *Definition condition*
  Let C,D be PRSs.
  Then the PRS B with conditions $\gamma_1, ..., \gamma_n, C := D$ is derivable under the premises Θ.

- *Quantifier and Implication condition*
  Let C,D be PRSs. Let $\Theta + FI(A) + FI(C) \vdash^P_{seq} FI(D)$.
  Then the PRS B with conditions $\gamma_1, ..., \gamma_n, C \rightarrow D$ is derivable under the premises Θ.

- *Assumption condition*
  Let C,D be PRSs and $\Theta + FI(A) + FI(C) \vdash^P_{seq} FI(D)$.
  Then the PRS B with conditions $\gamma_1, ..., \gamma_n, C => D$ is derivable under the premises Θ.

- *contradiction condition*
  Let $\Theta + FI(A) \vdash^P \bot$.
  Then the PRS B with conditions $\gamma_1, ..., \gamma_n, \mathit{contradiction}$ is derivable under the premises Θ.

- *Theorem PRS condition*
  Let B be a PRS, which is derivable under the premises $\Theta + FI(A)$,and G

be a PRS , which is derivable under the premises $\Theta + FI(A) + FI(B)$. Let T be the theorem PRS with goal G and body B.
Then the PRS C with conditions $\gamma_1, ..., \gamma_n, T$ is derivable under the premises Θ.

- *Lemma PRS condition*
  Let B be a PRS, which is derivable under the premises $\Theta + FI(A)$,and G be a PRS , which is derivable under the premises $\Theta + FI(A) + FI(B)$. Let L be the lemma PRS with goal G and body B.
  Then the PRS C with conditions $\gamma_1, ..., \gamma_n, L$ is derivable under the premises Θ.

Definition 4.2.7. *Naproche Acceptable PRS*
A PRS A is Naproche acceptable, if it is derivable under the premises $\langle\rangle$, the empty sequence.

We have defined two concepts which sound quite similar: The premises *of* a condition, and derivability *under* premises. This similarity is intentional. The premises *of* a PRS condition is the sequence which is used in the derivation of that PRS condition. For example, for a PRS A which is a condition in a Naproche acceptable PRS, the premises *of* A are the premises *under* which A needs to be derivable.

4.2.1 Natural Language and Proof Representation Structures

After having defined the sequent calculus in chapter 2, the Naproche language and general PRS in chapter 3, and the Naproche calculus for PRS in this chapter, we will now take a look at the connection of those ideas. We will consider questions like: Can every Naproche acceptable PRS be created by an appropriate discourse in the Naproche language? Can every derivable sequent be 'translated' into a PRS?

We use the PRS construction algorithm for the proofs in the section. Remember that 3.2.4 showed the contruction of a PRS by example. Chapter 5 and 6 in [13] have the details.

Lemma 4.2.1. *There are Naproche acceptable PRSs, such that those PRSs cannot be created by any discourse written in the Naproche language.*

Proof: Let A_0 be the empty PRS with no conditions or referents. Take A_1 as the PRS having only A_0 as condition. Then, we define A_2 as the PRS having only A_1 as condition. Recursively, we define A_{n+1} as the PRS which only has A_n as condition. By this construction, we can get PRS of arbitrary depth.

The empty discourse gets translated into the empty PRS A_0. But already A_1 cannot get created by a discourse in the Naproche language. The construction algorithm translates sentences in the Naproche language into PRS conditions. Such PRS conditions can, of course, be PRSs, but these PRSs are never empty. Therefore, all A_n with $n \geq 1$ cannot be produced by the Naproche system. *qed*

Note that this proof heavily depends on the PRS construction algorithm and the fact that it never creates empty PRSs as conditions. In upcoming versions of the Naproche system, the algorithm, as well as the PRS definitions, will most likely change, and with some consideration it should be possible to falsify the above lemma.

When we consider the relation between the the sequent calculus and PRSs we can prove some kind of completeness statement:

Lemma 4.2.2. *Let* $\Gamma \vdash \varphi$. *Let* $\Gamma = \langle \gamma_1, ..., \gamma_n \rangle$. *Then there is a PRS* A *such that*

$$FI(A) = \forall \text{free}(\gamma_1)\ \gamma_1 \rightarrow (\forall \text{free}(\gamma_2) - \text{free}(\gamma_1)\ \gamma_2 \rightarrow (...(\forall X\ \gamma_n \rightarrow \varphi)...))$$

We always quantify over the variables which are free in the current formula γ_i, *but not free in any* γ_j *for* $1 \leq j < i$. *So,* X *denotes all variables which are free in* γ_n, *but not free in any* γ_i *for* $1 \leq i < n$.

Proof: Let $\Gamma = \langle \gamma_1, ..., \gamma_n \rangle$. Then the discourse $\mathfrak{D}$ = [Let γ_1. ... Let γ_n. Then φ.] gets translated into a PRS with the required property. *qed*

Furthermore, if we only consider the formula image of a PRS A, then the Naproche language is expressive enough to find a discourse $\mathfrak{D}$, such that the to $\mathfrak{D}$ corresponding PRS has the same formula image as A. We get the following theorem:

Theorem 4.2.3. *Let* A *be a Naproche acceptable PRS. Then there is a discourse* $\mathfrak{D}$, *which, when parsed by the Naproche system, produces a PRS* B *for which holds* $FI(B) = FI(A)$.

Proof: Let A be a Naproche acceptable PRS with formula image $FI(A) = \langle \gamma_1, ..., \gamma_n \rangle$. Define $\mathfrak{D} = [\gamma_1....\gamma_n.]$, then $FI(B) = FI(A)$. *qed*

4.2.2 The Implementation of the Naproche Calculus

This diploma project was not only about the theoretical calculus, but also about its implementation. In this section, we show by example that the implementation and the calculus are in principle equivalent: A PRS is Naproche acceptable if the

implemented algorithm accepts it, and if a PRS gets accepted by the checking algorithm then it is Naproche acceptable.

The complete sourcecode can be found in the appendix A or, if you have the Naproche system installed, in the file *checker.pl*. The implementation was done by Dörthe Arndt and the author, with additional help of the trainees Bhoomija Ranjan and Shruti Gupta. Note that the code is written in the Prolog programming language. Let us look at the predicate *check_prs* first. The code is:

```
check_prs(PRS,Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger) :-
PRS = id~Id..conds~Conds,
!,

% discourse_to_prs gives us the conditions in the wrong order. Therefore we have to
% reverse them before we can check them.
reverse(Conds,Reversed_Conds),

% Check whether PRS is a structure PRS, in our case a lemma or a theorem
% e.g. check if Id = lemma.. or Id = theorem..
( ( atom_concat(theorem,_,Id) ; atom_concat(lemma,_,Id) ) ->
(
%theorem / lemma case
Reversed_Conds = [Goal,Proof],

% Get the Math IDs and the Premises of the Goal, but don't check
check_prs(Goal,Mid_begin,Goal_Mid_end,Premises_begin,Goal_Premises_end,nocheck),

% Check the Proof with updated Math IDs, discard Math IDs at the end
% Assumptions for Theorem must be made again!
check_prs(Proof,Goal_Mid_end,_,Premises_begin,Proof_Premises_end,Check_trigger),

% If Check_trigger = check see if Proof is a proof for Goal
( Check_trigger = check ->
( append(Premises_begin,Thm,Goal_Premises_end),
  fof_check(Proof_Premises_end,Thm,Id)
);
true
),

% Set New Math_Ids & Premises
Goal_Mid_end = Mid_end,
Premises_end = Goal_Premises_end

)
;

% If PRS is not a theorem just check the conditions
check_conditions(Id,Reversed_Conds,Mid_begin,Mid_end,Premises_begin,
  Premises_end,Check_trigger)
).
```

check_prs has six arguments, namely *PRS,Mid_begin,Mid_end,Premises_begin, Premises_end,Check_trigger*. *PRS* is the PRS which we want to check. *Mid_begin* and *Mid_end* store the math_id conditions which are accessible before and after parsing the PRS. *Premises_begin* and *Premises_end* are lists which contain the active premises, as first order formulas, before and after the parsing of the PRS, and *Check_trigger* is used to indicate whether we want to check the conditions of the PRS.

The programs first checks whether the input PRS is a normal, meaning non-structure, PRS or a theorem/lemma PRS.

```
% Check whether PRS is a structure PRS, in our case a lemma or a theorem
% e.g. check if Id = lemma.. or Id = theorem..
( ( atom_concat(theorem,_,Id) ; atom_concat(lemma,_,Id) ) ->
```

Let us first look at the simpler case of the input PRS being a non-structure PRS:

```
% If PRS is not a theorem just check the conditions
check_conditions(Id,Reversed_Conds,Mid_begin,Mid_end,Premises_begin,
  Premises_end,Check_trigger)
```

The checking algorithm accepts the PRS, if it accepts all the conditions of the PRS. When you look at the code of the check_conditions predicate, you see that they are checked sequentially. For a PRS with n conditions, the algorithm only tries to check the nth condition, after the first $n-1$ conditions got accepted. This corresponds to the general layout of our calculus: We add one condition after the other. If we want to derive a PRS with n conditions, then we first need to derive the PRS which contains the first $n-1$ conditions.

The second option in the check_prs predicate is that the input PRS is a structure PRS, meaning that it is a theorem or a lemma PRS.

```
%theorem / lemma case
Reversed_Conds = [Goal,Proof],

% Get the Math IDs and the Premises of the Goal, but don't check
check_prs(Goal,Mid_begin,Goal_Mid_end,Premises_begin,Goal_Premises_end,nocheck),

% Check the Proof with updated Math IDs, discard Math IDs at the end
% Assumptions for Theorem must be made again!
check_prs(Proof,Goal_Mid_end,_,Premises_begin,Proof_Premises_end,Check_trigger),

% If Check_trigger = check see if Proof is a proof for Goal
( Check_trigger = check ->
( append(Premises_begin,Thm,Goal_Premises_end),
  fof_check(Proof_Premises_end,Thm,Id)
);
true
),

% Set New Math_Ids & Premises
Goal_Mid_end = Mid_end,
Premises_end = Goal_Premises_end

)
```

The input PRS has two conditions. Both are PRSs, one is the goal and one is the proof.

```
%theorem / lemma case
Reversed_Conds = [Goal,Proof],
```

We run check_prs on the goal.

```
% Get the Math IDs and the Premises of the Goal, but don't check
check_prs(Goal,Mid_begin,Goal_Mid_end,Premises_begin,Goal_Premises_end,nocheck),
```

Note that the check_trigger parameter has *nocheck* as argument. Starting check_prs with *nocheck* basically gives the first order representation, the formula image, of the PRS. Here, the formula image of the PRS is added to the premises. The variable *Goal_Premises_end* contains a list which starts with the premises which we had at the start of this routine, *Premises_begin*, and ends with the formula image of the goal.

We then check the proof PRS under the premises which are stored in the list *Premises_begin*. After this is done, the variable *Proof_Premises_end* contains a list which starts with *Premises_begin* and ends with the formula image of the proof PRS.

```
% Check the Proof with updated Math IDs, discard Math IDs at the end
% Assumptions for Theorem must be made again!
check_prs(Proof,Goal_Mid_end,_,Premises_begin,Proof_Premises_end,Check_trigger),
```

If the check_trigger is set to check, we extract the formula image of the goal PRS from *Goal_Premises_end* and save it as *Thm*. Then we use the ATP to check whether *Proof_Premises_end* implies *Thm*.

```
( Check_trigger = check ->
( append(Premises_begin,Thm,Goal_Premises_end),
  fof_check(Proof_Premises_end,Thm,Id)
);
true
),
```

Let us compare this to the *Theorem PRS condition* from the Naproche calculus (*Lemma PRS condition* is analogue): The body PRS has to be derivable under the premises of the theorem PRS. This corresponds to the Naproche system being able to check the proof PRS using only *Premises_begin*. The second condition is that the goal PRS is derivable under the premises of the theorem PRS, plus the formula image of the body PRS. The line

```
fof_check(Proof_Premises_end,Thm,Id)
```

ensures exactly this. Only if these two conditions are fulfilled can we add a theorem PRS as condition in our calculus, and only if these two conditions are fulfilled does the Naproche system accept a theorem PRS.

For a second example, let us consider when the Naproche system accepts a negated PRS:

```
% ------------------------- Negation ----------------------------------------
% X = neg(PRS)
% Takes all premises from PRS and negates them.
% Check occurs after the negation.
check_conditions(Id,[neg(PRS) | Rest ],Mid_begin,Mid_end,Premises_begin,
  Premises_end,Check_trigger) :-
!,
% Get the Premises and Mids of PRS
```

```
% Check_trigger is nocheck as we want to prove the negates formulas!
check_prs(PRS,Mid_begin,New_Mid_begin,Premises_begin,Tmp_Premises_end,nocheck),

% Get the new premises and negate them
append(Premises_begin,Premises_PRS,Tmp_Premises_end),
negate_formulas(Premises_PRS,Negated_Premises_PRS),

% Check the Premises if Check_trigger = check
% otherwise just append them
( Check_trigger = check ->
fof_check(Premises_begin,Negated_Premises_PRS,Id);
true
),
append(Premises_begin,Negated_Premises_PRS,New_Premises_begin),

% Check the rest of the conditions with New_Mid_begin and New_Premises_begin
check_conditions(Id,Rest,New_Mid_begin,Mid_end,New_Premises_begin,
  Premises_end,Check_trigger).
```

Using the check_prs predicate with nocheck as argument, we get the formula image of the negated PRS:

```
% Get the Premises and Mids of PRS
% Check_trigger is nocheck as we want to prove the negates formulas!
check_prs(PRS,Mid_begin,New_Mid_begin,Premises_begin,Tmp_Premises_end,nocheck),
```

We use append to extract the formula image of the negated PRS and negate it.

```
% Get the new premises and negate them
append(Premises_begin,Premises_PRS,Tmp_Premises_end),
negate_formulas(Premises_PRS,Negated_Premises_PRS),
```

We check whether the ATP can prove the negated formula image from the premises:

```
% Check the Premises if Check_trigger = check
% otherwise just append them
( Check_trigger = check ->
fof_check(Premises_begin,Negated_Premises_PRS,Id);
true
),
```

And if that does succeed, we append the negated formula image to the premises we had so far, and then try to check the rest of the conditions:

```
append(Premises_begin,Negated_Premises_PRS,New_Premises_begin),

% Check the rest of the conditions with New_Mid_begin and New_Premises_begin
check_conditions(Id,Rest,New_Mid_begin,Mid_end,New_Premises_begin,
  Premises_end,Check_trigger).
```

This corresponds to the $\neg$*PRS condition* in the Naproche calculus: If we can prove the negation of the formula image from the premises, then we can add the negated PRS as condition.

We saw on two examples that the Naproche calculus and the implemented Naproche algorithm are equivalent. Apart for one exception, natural language quantification, one can similarly show the equality for the rest of the calculus. This exception a known bug and will be fixed in the next version.

4.3 Correctness and Completeness of the Naproche Calculus

For any calculus, we would like to know whether is is correct and/or complete. In our case, we have two problems. First, our PRS calculus heavily depends on the first order calculus we are using. And second, we have no way to check whether a definition is reasonable. However, if we restrict ourselves, we can get some results.

In the sequel, $\vdash^P$ means derivable in the calculus P, and $\vdash$ means derivable in the sequent calculus which we defined in in chapter 2. Remember that our sequent calculus is correct and complete. If P is complete, then $\Gamma \vdash \varphi$ implies $\Gamma \vdash^P \varphi$, and if P is correct, then $\Gamma \vdash^P \varphi$ implies $\Gamma \vdash \varphi$. We also get that if P is correct, then $\Gamma \vdash^P_{seq} \Theta$ implies $\Gamma \vdash_{seq} \Theta$.

Theorem 4.3.1. Completeness of the Naproche Calculus
If P is a complete calculus, then the Naproche Calculus is complete. More specifically, if $\Gamma \models \varphi$, then let A be the PRS which has one condition γ, which is an assumption condition with $\bigwedge \Gamma$ in the left-hand side PRS and φ in the right-hand side PRS. Then A is Naproche acceptable.

Proof: Obvious. *qed*

Theorem 4.3.2. Correctness of the Naproche Calculus
If P is a correct calculus, then the Naproche Calculus is correct. Meaning that if A is a Naproche acceptable PRS without definition conditions, then $\langle\rangle \vdash_{seq} FI(A)$.

Proof: We prove the stronger result that if A is derivable under the premises Θ, then $\Theta \vdash_{seq} FI(A)$, by induction over the depth of the PRS. This implies the theorem because every Naproche acceptable PRS is derivable under the empty sequence. Hence, for a Naproche acceptable PRS A we would get that $\langle\rangle \vdash_{seq} FI(A)$.

We begin the proof by considering empty PRSs, which have depth 0. For an empty PRS E, $FI(E) = \top$, and $\top$ is always derivable in our first order calculus (Lemma 2.3.5). So, assume that the result holds for all PRSs with depth $d \leq m - 1$, $m \geq 1$. For PRS with depth m, we prove the theorem by induction over the number of conditions.

Let A be a PRS, which is derivable under the premises Θ, and has depth $d(A) = m$. If A has no conditions, then A is the empty PRS and we are done. So, assume that A has conditions $\gamma_1, ..., \gamma_n$, and that the result holds for all PRS B with depth $d(B) \leq m$ and less than n conditions.

Because A is derivable under the premises Θ, the PRS A' with conditions $\gamma_1, ...\gamma_{n-1}$ is derivable under the premises Θ. All conditions of A' are also conditions of A, hence depth $d(A') \leq d(A) = m$. Using the induction hypothesis, we can conclude that $\Theta \vdash_{seq} FI(A')$. Remember that $FI(A) = FI(A') + \beta(\gamma_n)$. We consider the different options for the last condition γ_n.

$\gamma_n = holds(\varphi)$
Then $\Theta + FI(A') \vdash^P \varphi$, and since P is correct, $\Theta + FI(A') \vdash^P \varphi$ implies $\Theta + FI(A') \vdash \varphi$. Furthermore, $FI(A) = FI(A') + \langle\varphi\rangle$. Hence, we can use the last two statements to conclude $\Theta \vdash_{seq} FI(A)$.

$\gamma_n = B$, for a non-structure PRS B
B is derivable under the premises $\Theta + FI(A')$. Since $d(B) < m$, we can use the induction hypothesis and conclude $\Theta + FI(A') \vdash_{seq} FI(B)$. Therefore $\Theta \vdash_{seq} FI(A)$.

$\gamma_n = \neg B$, for a non-structure PRS B
By the calculus definition, $\Theta + FI(A') \vdash^P_{seq} \neg FI(B)$. Because P is correct, it follows that $\Theta + FI(A') \vdash_{seq} \neg FI(B)$. Thus, we get that $\Theta \vdash_{seq} FI(A)$.

$\gamma_n = C \to D$, for PRSs C, D
Since γ_n is a condition in A, we know that $d(C), d(D) < d(A) = m$. By the formula image definition, $\beta(C \to D) = \langle \forall\ free(\bigwedge FI(C)) - free(\bigwedge(\Theta + FI(A'))) \ \bigwedge FI(C) \to \varphi_i \mid \langle\varphi_i\rangle = FI(D)\rangle$, and by the calculus definition $\Theta + FI(A') + FI(C) \vdash^P_{seq} FI(D)$. Since P is correct, we can use Lemma 4.1.3 to conclude $\Theta + FI(A') \vdash_{seq} \langle \bigwedge FI(C) \to \varphi_i \mid \langle\varphi_i\rangle = FI(D)\rangle$. All variables in the set $free(\bigwedge FI(C)) - free(\bigwedge(\Theta + FI(A')))$ are not free in $\Theta + FI(A')$, and also not free in the sequent $\langle \forall\ free(\bigwedge FI(C)) - free(\bigwedge(\Theta + FI(A'))) \ \bigwedge FI(C) \to \varphi_i \mid \langle\varphi_i\rangle = FI(D)\rangle$. Therefore, we can use the $\forall$ Introduction rule in the sequent calculus to get $\Theta + FI(A') \vdash_{seq} \langle \forall\ free(\bigwedge FI(C)) - free(\bigwedge(\Theta + FI(A'))) \ \bigwedge FI(C) \to \varphi_i \mid \langle\varphi_i\rangle = FI(D)\rangle$. Therefore $\Theta \vdash_{seq} FI(A)$.

$\gamma_n = C => D$, for PRSs C, D
By the calculus definition, $\Theta + FI(A') + FI(C) \vdash^P_{seq} FI(D)$. We have two cases to consider. If the last condition of D is not *contradiction*, we proceed like in the implication case.

Else, we have that $\beta(C => D) = \langle \neg \bigwedge FI(C)\rangle$. Since the last condition of D is *contradiction*, and P is correct, we can apply lemma 4.1.1 to get $\Theta + FI(A') + FI(C) \vdash \bot$. We use lemma 4.1.3, to get $\Theta + FI(A') \vdash \bigwedge FI(C) \to \bot$. Now, a small calculation in the sequent calculus gives $\Theta + FI(A') \vdash \neg \bigwedge FI(C)$. Together, we conclude $\Theta \vdash_{seq} FI(A)$.

$\gamma_n = contradiction$
Since P is correct, $\Theta + FI(A') \vdash \bot$, and hence $\Theta \vdash_{seq} FI(A)$.

$\gamma_n = B$, for a theorem PRS B with goal G_B and body B_B
We have that $d(G_B), d(B_B) < d(A) = m$, and hence by induction and the calculus definition it follows that $\Theta + FI(A') \vdash_{seq} FI(B_B)$, $\Theta + FI(A') + FI(B_B) \vdash_{seq} FI(G_B)$. Putting these two together, we get $\Theta + FI(A') \vdash_{seq} FI(G_B)$ by lemma 4.1.4, and therefore $\Theta \vdash_{seq} FI(A)$.

$\gamma_n = B$, for a lemma PRS B with goal G_B and body B_B.
This case goes exactly like the theorem PRS case.

qed

Remark. Concerning definition conditions. Note that we did not cover definition conditions in the theorem. In their normal use, definitions are just abbreviations. These abbreviations extend our language and cannot be proven. At the moment, when encountering a definition, the Naproche system simply stores that definition as a premise, without checking it. In 5.2.2, we consider this problem in more detail.

Chapter 5

Practical Limitations and Future Work

So far, we have looked at the current implementation, as well as the mathematical, linguistic and computer scientific background of the Naproche system. This chapter is about its future.

We will first take a look at the practical limitation which the Naproche system has. Then, we will pick up a few points which we noted during the development of the current version, and which we think should be improved in the upcoming versions. The last section introduces the actual plans for the work on the Naproche system, starting from February 2009.

5.1 Practical Limitations of the Naproche System

The aim of the Naproche system is rather idealistic. A *natural language proof checker*, a program, which can process real natural language and deduce like, or better than, a human. Considering that even in normal conversations it is not always clear what the other person means, how can we teach a computer to understand us? In the section we will look at some practical limitation and how they affect our goal.

5.1.1 Correctness and Completeness

In the last chapter, we proved the theoretical correctness and completeness of the Naproche calculus. The one assumption we had for those proofs was that the calculus we use is complete and correct. In the Naproche system, this calculus comes from an ATP. If the ATP can derive a formula φ from some premises Γ, then φ is derivable from Γ in the calculus. Unfortunately, there is not one ATP

for which we know that it is correct and complete. Let us take a closer look at the problem, beginning with completeness.

The first thing to note is that some, if not most, ATPs are incomplete by design [25]. Furthermore, even if the ATP is complete in theory, there could still be errors somewhere in the code. Assuming that this is not the case, we still have the Naproche system, which could also have some bugs. Even if we would have a complete ATP system and no bugs in neither the ATP nor the Naproche system there is one more potential source for incompleteness: time. Naproche limits the time which the prover can take to discharge an obligation. Meaning, we could miss a solution just because we forced the ATP to stop too early.

While incompleteness is acceptable for an ATP, incorrectness is not. ATPs should be correct, but, of course, there could always be error in the program. Fortunately for us, there are two methods which can use to ensure that a program is doing what it is supposed to do. First, we can use software verification methods to check our code. Second, if software verification is too tedious or impractical, as [25] suggests, we can use the rather unmathematical concept of trust. Just by running enough examples, of which we know the correct answer, we can test the Naproche system/the ATP and, given that we do get the correct answers, establish some kind of trust in the system. For the Naproche system in particular, we can try the input on several ATPs and compare the results.

This leaves us with the time constraint problem, which is hard, if not impossible to avoid. If we want to maintain useability we need time constraints on the prover, which means that we do get incompleteness. (Unless the prover can instantly solve any given problem, which is not the case). In praxis, we could handle this problem by asking the user to change the proof in order to create easier obligations for the prover.

Altogether, we get the following: Using theoretical considerations for the foundations, software verification techniques for the implementation and different provers for comparison, we can be quite sure that a positive answer by the Naproche system, (*Proof Accepted*) means that the input is correct. If the Naproche system does not accept a given input, then this does not imply the incorrectness of the input. We can only conclude that the input is either incorrect, or that the tools we use are too weak to prove it. In other words, the Naproche system is incomplete, but we can hope for correctness. While this might seem discouraging at first, we should not forget that humans have the same limitation.

5.1.2 Natural Language

One of the goals of the Naproche system is *Natural Mathematical Language* as input format. At some point in the program, the input has to be translated into first (or higher) order logic. One problem which does arise here is that natural language cannot always be uniquely translated. There are cases, when several

translations are possible. For example *A and B implies C* could be read as $(A \wedge B) \rightarrow C$ or as $A \wedge (B \rightarrow C)$.

So far, Naproche, as well as its competitors Plato [32] and SAD [20], restrict their input and define unique translations into logic. Attempto [6] showed that one can get quite far with this approach. However, the best you can hope for is an input language which, when reading it, looks like natural English, but when you want to write a text for the program to check, you will have to learn the input language first.

Alternatively to the controlled language approach, one could allow ambiguities and try to work with them. This, unsurprisingly, leads to quite some problems. First of all, several translations mean more things to check. For example, if we take a text with n sentences, and every sentence has two possible translations, then, as a basic approach, we need to check 2^n translations for correctness, compared to only n if we do not allow ambiguity. Even if this exponential growth was acceptable, we would also need to redefine when a text is accepted. It is quite likely that not every possible translation gives the same result. When do we accept a text as correct? Only if every translation is accepted? Then the user would probably not be able to write a normal text. Only if there is a correct translation? This correct translation could be completely different from what the author intended to say. Then the program would say *Proof Accepted* even though what the author intended to say was wrong.

One way of dealing with text ambiguities is to ask the user which of the available options was meant. This, however, would move the Naproche away from automated proof checking, which is without user interaction, and towards proof assisting. If we want to go into this direction, we should take a closer look at Coq [5] and Isabelle [18].

In my personal opinion, we should stay with the controlled language concept for the time being. Attempto managed to make a part of a language, English, unambiguous (The current vocabulary file has close to 100.000 entries [19]). The mathematical language is a tiny subset of English, and furthermore, it is already quite precise in its meaning. The controlled language would probably be not too far away from what mathematicians actually use, and therefore be easy to learn and teach. The Naproche project, as a group of linguists and mathematicians, should be able to get a lot more out of the concept of a restricted language for mathematics, than what is currently possible.

5.2 Possible Improvements

What follows is a list of things we noticed in the development of the last version of the Naproche system, and which we would like to change for upcoming versions.

5.2.1 Usability and Communication

At the moment, using Naproche is far from simple. Even if we consider, that it is a scientific project which is still in development, there is a lot we can improve.

Installation

To get Naproche up and running, one needs to install more than 5 different programs and even change a few lines in the code. On top of that, not one person knows all the things which one needs to consider, when installing it. Setting up a decent installation routine is a must for future versions.

Documentation

While the sourcecode documentation is quite good, the end user documentation is almost non-existent. The input language is only partially defined, we have no tutorials, nothing which helps a end-user.

Website

Nowadays, the internet is the primary source of information. The Naproche Website is both outdated and rudimentary. We need up to date content, the possibility to download recent and older versions, as well as pdfs of all papers about Naproche.

I propose, that we schedule cycles, for example every 6 month, in which we release updated versions of the Naproche system. These versions are released on the website, complete with a working installer as well as proper documentation for the end user.

5.2.2 Proving

The actual proving process of the Naproche system is very underdeveloped. We already mentioned a few issues in the last chapter. The following are ideas for improving the capabilities of the Naproche system.

Definitions

The Naproche system treats definition sentences as equivalence statements. While this can already cause logic problems, we also might get problems with the pure amount of definitions. We will consider the logic problems first:

In order to check that conditions do make sense, there are a few things we can do. One idea is the following algorithm: First, we check whether what we define is a new word. If it is then we have a definition which is simply an abbreviation.

If we do not define a new word, but instead use one from the vocabulary, we check if there is already a definition associated with that word. If yes, we check for equivalence of those two definitions. If they are equivalent, everything is fine. If not we should report an error. If there is no definition for that word then it is ok to create one.

Alternatively, definitions could be treated as additional assumptions. Changing the semantics to "This proof is correct under these definitions". One drawback of this approach is that the formulas that the ATP has to work with would get even longer.

For handling definitions, Andriy Paskevych developed the method of definition expanding [15]. Instead of saving definitions as normal premises, they are stored separately. The prover gets the text without the definitions first, and only if it cannot succeed without the definitions does the program provide him with more information. This way, we can simplify the problem for the prover, which in return increases the proving capabilities of our program.

Proving on PRS Level

In the calculus, we defined that the formula image of an assumption ended by a contradiction would be the negated assumption. This is already some kind of proving on PRS level.

Another proof type, which could be implemented at this level would be proof by induction or recursions. Generally, a calculus which can be changed while running the program is thinkable. The user could define when something is correct. For example, we could have a rule in the calculus which states that when a statement for the natural numbers holds for 0 and every successor, then it holds for all natural numbers.

But as one should prove that additions to the calculus are correct, we might as well just leave everything to the ATP. We need to do more research to see whether such an option would be useful.

Premises

Eventually, we will have to worry about the power of the underlying prover of the Naproche system. So far, we did not have any problems but we did not work with bigger examples either. The more premises you give an ATP, the worse it will preform. The way we handle premises now we will get proof obligations with dozens of premises, most of which are unnecessary for the actual proof. In order to be able to check bigger texts, we need to find a way to keep the amount of premises for each obligation small.

Mizar [16] handles this problem by explicitly asking for the appropriate theorem for each proof step. Unless you know the entire library by heart, this makes

writing proofs in Mizar quite tedious. So we do not want to use this concept for a natural language proof checker.

One idea we had, was the development of some kind of metric for statements. This metric would continuously be updated. When the prover tries to discharge an obligation, the ATP first tries to prove it, using only those statements which are *near*, according to the metric, the current obligation. If it is not successful, the algorithm would slowly increase the allowed *distance* which a statement can have from the obligation, in order to be considered for the discharge. This way, we can keep the number of the premises which are used for the proof small. The method of definition expanding, which we mentioned earlier, would also nicely fit into this concept.

Such a metric could be based on

- The actual distance in the input text.
- The kind of statement (Theorem/Lemma/Proof Statement).
- The frequency of how often the statement was used in previous proofs.
- Explicit reference in the text (e.g. by lemma n ...).

For a useful definition, we will probably need not only more theory, but also lots of tests on different texts. If we do want to keep the proving part of the Naproche system in Bonn, I strongly recommend that we do more research in this area.

5.2.3 XML

In the current version of Naproche, the conversion from the input text to XML does little more than adding a unique number to each sentence. If we do want to keep the approach of converting to XML first, we should think about getting more information from the texmacs file, and, subsequentially, having more information in the XML. Furthermore, it should be possible for the writer to supply additional information for Naproche while writing the document. This could also be saved in the XML.

Either way, it would be necessary to redesign the XML specifications. For this, we should take a closer look at Michael Kohlhase and his group. They are working on a semantic markup format for mathematical documents: OMDoc [12]. This format could replace our current XML standard. It can handle all the things we need from an XML format right now, and, additionally, it allows the user to give more information, which in turn could be used in the creation of the PRS and the proving of the text.

Another benefit of OMDoc would be that we would allow a broader input. The user would not be forced to use Texmacs. As long as his text is convertible

into OMDoc, it can be checked by Naproche. This could make Naproche more appealing to users, which will be more and more important as the development progresses.

5.3 The Future of Naproche

After all these considerations of possible improvements, what is the actual plan for the future of Naproche? At the *Conference on Intelligent Computer Mathematics*, we had the opportunity to meet the VeriMathDoc group which is based in Saarbrücken and Bremen. It became obvious that we have very similar ideas and concepts, and that we therefore could both benefit from a cooperation. VeriMathDoc is particularly strong in all IT aspects, be it theorem proving, programming or program interaction. Naproche, on the other hand, has its focus on natural language and mathematics.

The plan is that, starting from autumn 2008, the VeriMathDoc and Naproche teams will work together. Each team will continue to work on its own project, but we will share the results of our respective studies as well as the programs which can be used by both teams.

Chapter 6

Conclusion

In this thesis, we introduced the Naproche project, explained the Naproche system in greater detail, and talked about the its limitations. We showed that with relatively simple means, namely the current Naproche language and the current Naproche system, it is already possible to formulate and proof such non-trivial theorems like the Burali-Forti paradox. Such early results suggest that with further research, it should be easily possible to significantly improve both the Naproche language as well as the Naproche system.

Even though we are, to our knowledge, so far the only mathematical-linguistic group which is working in this area, the public reception of the Naproche project was very good. Conferences, exhibitions like the "Wissenschaftszelt", and our Naproche seminar gave us the opportunity to talk to scientists from all three involved areas, mathematics, linguistics and computer science. Most of them seemed very interested in our work, and were eager for hear about new results.

In conclusion, the author argues that the Naproche project is very promising and should be pursued further. Not only are more and better results feasible, but also asked for by the scientific community.

Appendix A

The Sourcecode of checker.pl

```
 :-module(check_prs,[check_prs/6]).


:- use_module(library(pldoc)).

:- ensure_loaded(naproche(gulp4swi)).
:- use_module('premises').
:- use_module('fof_check').
:- use_module(naproche(prs)).

% assumption marker
:- op(901, xfx, user:(=>)).

% implication marker
:- op(901, xfx, user:(==>)).

% definition marker
:- op(901, xfx, user:(:=)).

PRS_A := PRS_B :-
(is_prs(PRS_A);is_neg_prs(PRS_A)),
(is_prs(PRS_B);is_neg_prs(PRS_B)).

PRS_A => PRS_B :-
(is_prs(PRS_A);is_neg_prs(PRS_A)),
(is_prs(PRS_B);is_neg_prs(PRS_B)).

PRS_A ==> PRS_B :-
(is_prs(PRS_A);is_neg_prs(PRS_A)),
(is_prs(PRS_B);is_neg_prs(PRS_B)).

/** <module> High level proof checking predicate
 *
 *  This module provides predicates to check a PRS using a first order prover.
 *
 */



%% check_prs(+PRS:prs,+Mid_begin:list,-Mid_end:list,+Premises_begin:list(DOBSOD),
%% -Premises_end:list(DOBSOD),+Check_trigger:(check | nocheck)) is det.
%
% True if PRS is logically valid.
%
% Depending on the Check_trigger, the PRS is translated into TPTP FOL and checked,
```

```
% using the prover Prover if Check_trigger = check, or just translated into TPTP FOL
% if Check_trigger = nocheck.
% A PRS is logically valid, if all its conditions are logically valid.
%
% NOTE: In order to actually check the input you need to define the following predicates:
% check_time(Time)
% check_prover(Checker)
% check_size(Outputsize)
% These are normally defines in load.pl. fof_check will not work without them!
%
% @param PRS is the PRS to be checked.
% @param Mid_begin is the list of available Math_IDs before the start of the predicate.
% @param Mid_end is the list of available Math_IDs at the end of the predicate
% @param Premises_begin is the list of premises from which we try to prove the PRS.
% @param Premises_end is the list containing Premises_begin and every formula we
%  proved through the PRS.
% @param Check trigger gives us the option to either try to prove every formula we
%  encounter ( check ) or to just go through the structure of the proof without
%  running a prover (nocheck).
%
%
% @tbd Mid handling in Negation case. Should they be negated?

check_prs(PRS,Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger) :-
PRS = id~Id..conds~Conds,
!,

% discourse_to_prs gives us the conditions in the wrong order. Therefore we have to
% reverse them before we can check them.
reverse(Conds,Reversed_Conds),

% Check whether PRS is a structure PRS, in our case a lemma or a theorem
% e.g. check if Id = lemma.. or Id = theorem..
( ( atom_concat(theorem,_,Id) ; atom_concat(lemma,_,Id) ) ->
(
%theorem / lemma case
Reversed_Conds = [Goal,Proof],

% Get the Math IDs and the Premises of the Goal, but don't check
check_prs(Goal,Mid_begin,Goal_Mid_end,Premises_begin,Goal_Premises_end,nocheck),

% Check the Proof with updated Math IDs, discard Math IDs at the end
% Assumptions for Theorem must be made again!
check_prs(Proof,Goal_Mid_end,_,Premises_begin,Proof_Premises_end,Check_trigger),

% If Check_trigger = check see if Proof is a proof for Goal
( Check_trigger = check ->
( append(Premises_begin,Thm,Goal_Premises_end),
  fof_check(Proof_Premises_end,Thm,Id)
);
true
),

% Set New Math_Ids & Premises
Goal_Mid_end = Mid_end,
Premises_end = Goal_Premises_end

)
;

% If PRS is not a theorem just check the conditions
check_conditions(Id,Reversed_Conds,Mid_begin,Mid_end,Premises_begin,
  Premises_end,Check_trigger)
).

% ---------------------------------- check_neg_prs --------------------------------------


%% check_neg_prs(+NEG_PRS:neg(prs),+Mid_begin:list,-Mid_end:list,
%% +Premises_begin:list(DOBSOD),-Premises_end:list(DOBSOD),
```

```
%% +Check_trigger:(check | nocheck)) is det.
%
% True if NEG_PRS is logically valid, takes a negated PRS as input.
%
% Depending on the Check_trigger, the NEG_PRS is translated into TPTP FOL and checked,
% using the prover Prover if Check_trigger = check, or just translated into TPTP FOL
% if Check_trigger = nocheck.
% A NEG_PRS is logically valid, if all its conditions are logically valid.
%
% NOTE: In order to actually check the input you need to define the following predicates:
% check_time(Time)
% check_prover(Checker)
% check_size(Outputsize)
% These are normally defines in load.pl. fof_check will not work without them!
%
% @param NEG_PRS is the negated PRS to be checked.
% @param Mid_begin is the list of available Math_IDs before the start of the predicate.
% @param Mid_end is the list of available Math_IDs at the end of the predicate
% @param Premises_begin is the list of premises from which we try to prove the PRS.
% @param Premises_end is the list containing Premises_begin and every formula we
%  proved through the PRS.
% @param Check trigger gives us the option to either try to prove every formula we
%  encounter ( check ) or to just go through the structure of the proof without
%  running a prover (nocheck).


check_neg_prs(neg(PRS),Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger) :-
!,

% Check the PRS
check_prs(PRS,Mid_begin,Mid_end,Premises_begin,PRS_Premises_end,Check_trigger),

% Get the premises which are added by the PRS
append(Premises_begin,PRS_Premises,PRS_Premises_end),

% Negate the PRS_Premises
negate_formulas(PRS_Premises,Negated_Premises),

% Append the negated Premises
append(Premises_begin,Negated_Premises,Premises_end).



% ---------------------------------- check_conditions -----------------------------

%% check_conditions(+Id:atom,+Conditions:list,+Mid_begin:list,-Mid_end:list,
%% +Premises_begin:list(DOBSOD),-Premises_end:list(DOBSOD),
%% +Check_trigger:(check|nocheck)) is det.
%
% True if every Element of the List is logically valid, or if the list is empty
%
% check_conditions checks the Elements of the List one ofter the other.
%
% Possible Conditions X are:
%
% * X is a PRS
% * X = math_id(_,_)
% * X = holds(Y)
% * X = PRS_A := PRS_B Definition
% * X = PRS_A => PRS_B Assumption
% * X = neg(PRS) Negation
% * X = PRS_A ==> PRS_B For all
% * X = PRS_A ==> PRS_B Implication
% * X = contradiction Contradiction
%
% NOTE: In order to actually check the input you need to define the following predicates:
% check_time(Time)
% check_prover(Checker)
% check_size(Outputsize)
% These are normally defines in load.pl. fof_check will not work without them!
```

```
%
% @param Id is the ID of the PRS we are checking
% @param Conditions is the list of conditions which we try to prove.
% @param Mid_begin is the list of available Math_IDs before the start of
%   the predicate.
% @param Mid_end is the list of available Math_IDs at the end of the predicate
% @param Premises_begin is the list of premises from which we try to prove the PRS.
% @param Premises_end is the list containing Premises_begin and every formula
%   we proved through the PRS.
% @param Check trigger gives us the option to either try to prove every formula
%   we encounter ( check ) or to just go through the structure of the proof
%   without running a prover (nocheck).


% ------------------------- Empty List ----------------------------------------
% Empty list is valid.
% Mid_begin = Mid_end
% Premises_begin = Premises_end
check_conditions(_,[],Mid_begin,Mid_begin,Premises_begin,Premises_begin,_) :- !.


% ------------------------- PRS -----------------------------------------
% X is a PRS
% Check PRS then proceed with the updated Math_IDs and Premises.
check_conditions(Id,[X|Rest],Mid_begin,Mid_end,Premises_begin,
  Premises_end,Check_trigger) :-
is_prs(X),
!,
check_prs(X,Mid_begin,X_Mid_end,Premises_begin,X_Premises_end,Check_trigger),

% Use the updated Mid and Premises for the remaining check.
check_conditions(Id,Rest,X_Mid_end,Mid_end,X_Premises_end,Premises_end,Check_trigger).


% ------------------------- math_id(_,_) ----------------------------------------
% X = math_id(_,_)
% Update Mid_begin and proceed
check_conditions(Id,[X|Rest],Mid_begin,Mid_end,Premises_begin,
  Premises_end,Check_trigger) :-
X = math_id(_,_),
!,
append(Mid_begin,[X],New_Mid_begin),
check_conditions(Id,Rest,New_Mid_begin,Mid_end,Premises_begin,Premises_end,Check_trigger).

% ------------------------- holds(_) ----------------------------------------
% X = holds(Y)
% We use the Prover to check whether Y follows from the premises.
% If yes, then the TPTP representation of Y is added to the premises, else
% we throw an error.
% Check_trigger specifies whether we do the actual check or just update the premises.
check_conditions(Id,[ holds(Y) |Rest],Mid_begin,Mid_end,Premises_begin,
  Premises_end,Check_trigger) :-
!,
% Find the Formula corresponding to Y.
% If it can't be found in Mid_begin throw an error
Z = math_id(Y,Formula_PRS),
member(Z,Mid_begin),
% If FAIL THROW ERROR !!!

% Get rid of the math(..) part.
Formula_PRS = math(Formula_FOL),
(((Formula_FOL = type~quantifier),!);
 ((Formula_FOL = type~relation),!);
         ((Formula_FOL = type~logical_symbol),!)),!,

% If Check_trigger = check use Prover to check whether Formula follows from the premises
% If it can't be proven throw an error.
% If Check_trigger = nocheck do nothing.
( Check_trigger = nocheck -> true
; fof_check(Premises_begin,[Formula_FOL],Id)
```

```
),
% If FAIL, THROW ERROR!!!

% Update premises
append(Premises_begin,[Formula_FOL],New_Premises_begin),

!,
check_conditions(Id,Rest,Mid_begin,Mid_end,New_Premises_begin,Premises_end,Check_trigger).


% ------------------------- Definition ---------------------------------------
% X = A := B
% A and B both can be either a PRS or a negated PRS.
% A Definition is just one more premise. Therefore we update the premises and proceed.
check_conditions(Id,[ X |Rest],Mid_begin,Mid_end,Premises_begin,
  Premises_end,Check_trigger) :-
X = (A:=B),
!,

% Extract the list of Premises from A and B
( is_prs(A) ->
 check_prs(A,Mid_begin,A_Mid_end,Premises_begin,A_Premises_end,nocheck);
 check_neg_prs(A,Mid_begin,A_Mid_end,Premises_begin,A_Premises_end,nocheck)
),
append(Premises_begin,List_A,A_Premises_end),!,

( is_prs(B) ->
 check_prs(B,A_Mid_end,_,A_Premises_end,B_Premises_end,nocheck);
 check_neg_prs(B,A_Mid_end,_,A_Premises_end,B_Premises_end,nocheck)
),
append(A_Premises_end,List_B,B_Premises_end),!,

% Update the premises
update_definitions(Premises_begin,New_Premises_begin,List_A,List_B),
!,
check_conditions(Id,Rest,Mid_begin,Mid_end,New_Premises_begin,Premises_end,Check_trigger).


% ------------------------- Assumption ---------------------------------------
% X = A => B
% A and B both can be either a PRS or a negated PRS.
% This results in a premises update:
% For all premises X from B we add
% ! [Free variables in A] : Fol_A -> X
% to the list of premises available
check_conditions(Id,[ X |Rest],Mid_begin,Mid_end,Premises_begin,
  Premises_end,Check_trigger) :-
X = (A=>B),
!,

% Check A. The values of Mid_tmp and Premises_tmp will be given to
% B as _begin values.
% We use nocheck as these are Assumptions and need not be checked.
( is_prs(A) ->
 check_prs(A,Mid_begin,Mid_tmp,Premises_begin,A_Premises_end,nocheck);
 check_neg_prs(A,Mid_begin,Mid_tmp,Premises_begin,A_Premises_end,nocheck)
),!,

% Extract the list of premises from A
append(Premises_begin,List_A,A_Premises_end),


% Check B with all the Premises and Mid from A included
% The Mid_begin values are Mid_tmp and the Premises_begin values are Premises_tmpa,
% both of which we got from check_prs(A,..)
% The Mid_end value is not important ( ?? )
% The Premises_end value will later be used for all-quantification.
%
% If Check_trigger = nocheck then don't check B, else do check it
( is_prs(B) ->
```

```
 ( Check_trigger = nocheck ->
check_prs(B,Mid_tmp,_,A_Premises_end,B_Premises_end,nocheck);
check_prs(B,Mid_tmp,_,A_Premises_end,B_Premises_end,check)
 );
 ( Check_trigger = nocheck ->
check_neg_prs(B,Mid_tmp,_,A_Premises_end,B_Premises_end,nocheck);
check_neg_prs(B,Mid_tmp,_,A_Premises_end,B_Premises_end,check)
 )
),
!,

% Two cases:
% A : We deal with a normal assumption
% B : The user made a proof by contradiction
% The last entry in B_Premises_end determines that
( append(_,[type~relation..arity~0..name~'$false'],B_Premises_end) ->
(
% Contradiction case
% Negate Assumption and proceed
negate_formulas(List_A,Negated_A),
append(Premises_begin,Negated_A,New_Premises_begin)
)
;
(
% For all Formulas X in Premises_tmpb / Premises_tmp_a add
% a new over all free variables in A quantified Statement of the form
% ! [Free variables in A] : Fol_A -> X
% to our Premises storage
append(A_Premises_end,Premises_B,B_Premises_end),
update_assumption(Premises_begin,New_Premises_begin,List_A,Premises_B)
)
),

% Reset the Mid values to before the Assumption
% The Premises_begin becomes the New_Premises_begin which we got
% from the last predicate.
!,
check_conditions(Id,Rest,Mid_tmp,Mid_end,New_Premises_begin,Premises_end,Check_trigger).


% ------------------------- Negation ---------------------------------------
% X = neg(PRS)
% Takes all premises from PRS and negates them.
% Check occurs after the negation.
check_conditions(Id,[neg(PRS) | Rest ],Mid_begin,Mid_end,Premises_begin,
  Premises_end,Check_trigger) :-
!,
% Get the Premises and Mids of PRS
% Check_trigger is nocheck as we want to prove the negates formulas!
check_prs(PRS,Mid_begin,New_Mid_begin,Premises_begin,Tmp_Premises_end,nocheck),

% Get the new premises and negate them
append(Premises_begin,Premises_PRS,Tmp_Premises_end),
negate_formulas(Premises_PRS,Negated_Premises_PRS),

% Check the Premises if Check_trigger = check
% otherwise just append them
( Check_trigger = check ->
fof_check(Premises_begin,Negated_Premises_PRS,Id);
true
),
append(Premises_begin,Negated_Premises_PRS,New_Premises_begin),

% Check the rest of the conditions with New_Mid_begin and New_Premises_begin
check_conditions(Id,Rest,New_Mid_begin,Mid_end,New_Premises_begin,Premises_end,
  Check_trigger).

% ------------------------- Implication & for all/exists --------------------------------
% X = A ==> B
% A and B both can be either a PRS or a negated PRS.
```

```
% Find the Formulas in PRS_A and adds them as Premises. Then proceeds to check PRS_B
% with the updated premises
check_conditions(Id,[ X | Rest ],Mid_begin,Mid_end,Premises_begin,
  Premises_end,Check_trigger) :-
X = (A==>B),
!,

% Get the Premises and Mids from A
% We don't check as these are Assumptions!
( is_prs(A) ->
check_prs(A,Mid_begin,A_Mid_begin,Premises_begin,A_Premises_end,nocheck);
check_neg_prs(A,Mid_begin,A_Mid_begin,Premises_begin,A_Premises_end,nocheck)
),!,

% Check B with the updated Premises
% The Math IDs of A and B will not matter for the Rest of the proof.
( is_prs(B) ->
check_prs(B,A_Mid_begin,_,A_Premises_end,B_Premises_end,Check_trigger);
check_neg_prs(B,A_Mid_begin,_,A_Premises_end,B_Premises_end,Check_trigger)
),!,

% Get the Formulas in A and B
append(Premises_begin,List_A,A_Premises_end),
append(A_Premises_end,List_B,B_Premises_end),

% Update the original premises:
% For each formula F in B add
% ! [Free A] : A => F
% to the premises.
( List_A = [] ->
(
% A comes from a for all statement in natural language
% We quantify over each Mref of A
A = mrefs~Mrefs_A..id~Id_A,
 (atom_concat('prefix_forall',_,Id_A) ->
  update_for_all(Premises_begin,New_Premises_begin,Mrefs_A,List_B);
             update_there_exists(Premises_begin,New_Premises_begin,Mrefs_A,List_B)
 )
) ;
% A comes from a normal Implication
update_implication(Premises_begin,New_Premises_begin,List_A,List_B)
),

% Check the rest of the conditions with Mid_begin and New_Premises_begin
check_conditions(Id,Rest,Mid_begin,Mid_end,New_Premises_begin,Premises_end,Check_trigger).


% -------------------------- contradiction -----------------------------------------
check_conditions(Id,[contradiction|Rest],Mid_begin,Mid_end,Premises_begin,
  Premises_end,Check_trigger) :-
!,

% Check whether we can conclude $false from the premises so far.
( Check_trigger = check ->
fof_check(Premises_begin,[type~relation..arity~0..name~'$false'],Id);
true
),

% Add [contradiction] to the premises
append(Premises_begin,[type~relation..arity~0..name~'$false'],New_Premises_begin),

% Check the rest of the conditions with New_Premises_begin
check_conditions(Id,Rest,Mid_begin,Mid_end,New_Premises_begin,Premises_end,Check_trigger).


% ------------------------- Everything else ---------------------------------------
% Every other case
%check_conditions([_|Rest],Mid_begin,Mid_end,Premises_begin,Premises_end) :-
% !,
% % throw error: 'Unknown condition'
```

```
% check_conditions(Rest,Mid_begin,Mid_end,Premises_begin,Premises_end). % or fail?
```

Appendix B

The math lexicon

```
:- module(math_lexicon,[math_lexicon/2]).

/**     <module>  This module contains all mathematical items which math_grammar can parse
*
*
*/


%%      math_lexicon(?Item:list(int),DOBSOD:gulp_list)
%
%       math_lexicon lists all items and their properties which math_grammar can parse.
%
%       ==
%       math_lexicon("x",type~variable..arity~0..name~'x').
%       math_lexicon("f",type~function..arity~1..name~'f').
%       ==
%
%       @param Item     The Name of the item as a String.
%       @param DOBSOD   The DOBSOD representation of the Item.
%


% Variables
math_lexicon("t",type~variable..arity~0..name~'t').
math_lexicon("u",type~variable..arity~0..name~'u').
math_lexicon("v",type~variable..arity~0..name~'v').
math_lexicon("w",type~variable..arity~0..name~'w').
math_lexicon("x",type~variable..arity~0..name~'x').
math_lexicon("y",type~variable..arity~0..name~'y').
math_lexicon("z",type~variable..arity~0..name~'z').

% Constants
math_lexicon("\u2205",type~constant..arity~0..name~'emptyset').
math_lexicon("c",type~constant..arity~0..name~'c').

% Functions
math_lexicon("f",type~function..arity~1..name~'f').
math_lexicon("f2",type~function..arity~1..name~'f2').
math_lexicon("g",type~function..arity~2..name~'g').
math_lexicon("h",type~function..arity~3..name~'h').

% Relation
math_lexicon("contradiction",type~relation..arity~0..name~'$false').
math_lexicon("Ord",type~relation..arity~1..name~'ord').
math_lexicon("Trans",type~relation..arity~1..name~'trans').
math_lexicon("R",type~relation..arity~2..name~'r').
```

```
math_lexicon("=",type~relation..arity~2..name~'=').
math_lexicon("\u2260",type~relation..arity~2..name~'~=').
math_lexicon("\u2208",type~relation..arity~2..name~'in').
math_lexicon("\u2264",type~relation..arity~2..name~'leq').
math_lexicon("\u2265",type~relation..arity~2..name~'geq').
math_lexicon("\u003C",type~relation..arity~2..name~'less').
math_lexicon("\u003E",type~relation..arity~2..name~'greater').


% logical symbols
math_lexicon("\u00ac",type~logical_symbol..arity~1..name~'~').
math_lexicon("\u2227",type~logical_symbol..arity~2..name~'&').
math_lexicon("\u2228",type~logical_symbol..arity~2..name~'|').
math_lexicon("\u2192",type~logical_symbol..arity~2..name~'=>').
math_lexicon("\u2194",type~logical_symbol..arity~2..name~'<=>').


% quantifiers
math_lexicon("\u2200",type~quantifier..arity~2..name~'!').
math_lexicon("\u2203",type~quantifier..arity~2..name~'?').
```

Appendix C

The Naproche Language

```
<text>              ::= [<open-assumption>]*,
                        [<definition>]*,
                        [<lemma> |
                        <theorem>]*;

<theorem>           ::= "theorem.",
                        [<definition>]*,
                        [<assumption>]*,
                        [<ext-statement>]+,
                        "proof.",
                        [<assumption> |
                        <ext-statement> ]*,
                        [<ext-statement>]+,
                        "qed.";


<lemma>             ::= "lemma.",
                        [<definition>]*,
                        [<assumption>]*,
                        [<ext-statement>]+,
                        "proof.",
                        [<assumption> |
                        <ext-statement> ]*,
                        [<ext-statement>]+,
                        "qed.";

<open-assumption>   ::= <assumption>;

<closed-assumption> ::= <assumption>,
                        [<ext-statement> |
                        <lemma> ]*,
                        <assumption-close>;

<definition>        ::= "define",
                        <sub-statement>,
                        ["iff" | "if and only if"],
                        <sub-statement>.

<assumption>        ::= [ "assume that" |
                        "assume for a contradiction that" |
                        "let" |
                        "consider" ],
                        <simple-statement>;

<assumption-close>  ::= "thus",
```

```
                        <simple-statement>;

<ext-statement>     ::= [ <uni-quant> |
                        <ext-quant> |
                        <implication> |
                        <negation> |
<statement> |
                        <closed-assumption> ]+;

<simple-statement>  ::= [ <uni-quant> |
                        <ext-quant> |
                        <implication> |
                        <negation>
                        <sub-statement>]+;

<uni-quant>         ::= "for all",
                        <variable>,
                        ","
                        [<simple-statement>]+;

<ext-quant>         ::= "there is an",
                        <variable>,
                        "such that",
                        <simple-statement>;

<implication>       ::= <sub-statement>,
                        "implies",
                        <sub-statement>;

<negation>          ::= "not",
                        <sub-statement>;

<statement>         ::= [ "then" |
                        "hence" |
                        "recall that" |
                        "but" |
                        "in particular" |
                        "observe that" |
                        "together we have" |
                        "so" ],
                        <sub-statement>;

<sub-statement>     ::= A formula in the underlying first order language

<variable>          ::= A variable of the underlying first order language
```

List of Abbreviations

Abbreviation	Description
ATP	Automated Theorem Prover
DRS	Discourse Representation Structure
DRT	Discourse Representation Theory
PRS	Proof Representation Structure
TPTP	Thousands of Problems for Theorem Provers
Dref	Discourse Referent
Mref	Mathematical Referent
Rref	Textual Referent

List of Figures

Bibliography

[1] P. Blackburn and J. Bos. Representation and Inference for Natural Language: Working with Discourse Representation Structures. 1999.

[2] Cesare Burali-Forti. Una questione sui numeri transfiniti. *Rendiconti del Circolo Matematico di Palermo*, 11:154–164, 1897.

[3] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik*. Spektrum, 2007.

[4] Texmacs Editor. URL: www.texmacs.org/.

[5] Jean-Christophe Filliatre, Hugo Herbelin, Bruno Barras, Bruno Barras, Samuel Boutin, Samuel Boutin, Cristina Cornes, Cristina Cornes, Judicael Courant, Judicael Courant, Chetan Murthy, Chetan Murthy, Catherine Parent, Catherine Parent, Christine Paulin-mohring, Christine Paulin-mohring, Amokrane Saibi, Amokrane Saibi, Benjamin Werner, and Benjamin Werner. The Coq Proof Assistant Reference Manual. Technical report, 1998.

[6] Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Discourse Representation Structures for ACE 6.0. Technical report, University of Zurich, 2008.

[7] K. Gödel. Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatsheft für Mathematik und Physik*, 37, 1930.

[8] L. Henkin. The Completeness of the First-Order Functional Calculus. *The Journal of Symbolic Logic*, 14, 1949.

[9] Andrew Irvine. Russell's Paradox. In *The Stanford Encyclopedia of Philosophy*. Fall 2008.

[10] H. Kamp and U. Reyle. *From Discourse to Logic: Introduction to Model-theoretic Semantics of Natural Languge*. Kluwer Academic Publisher, 1993.

[11] Peter Koepke. Mathematical Logic. An Introduction. 2006.

[12] Michael Kohlhase. *OMDoc - An Open Markup Format for Mathematical Documents [version 1.2]*, volume 4180 of *Lecture Notes in Computer Science*. Springer, 2006.

[13] Nickolay Kolev. Generating Proof Representation Structures for the Project NAPROCHE. Master's thesis, University of Bonn, 2008.

[14] Daniel Kühlwein. A short introduction to Naproche. 2008.

[15] Alexander Lyaletski, Andrei Paskevich, and Konstantin Verchinine. The SAD system: Deductive assistance in an intelligent linguistic environment. In *Intelligent Systems*, 2006.

[16] Roman Matuszewski and Piotr Rudnicki. Mizar: the first 30 years. *Mechanized Mathematics and Its Applications*, 4:2005, 2005.

[17] John Mccarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[18] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[19] Kaarel Kaljurand Norbert E. Fuchs. *The Language ACE.* University of Zurich, 2006.

[20] Andriy Paskevych. *Méthodes de formalisation des connaissances et des raisonnements mathématiques: aspects appliqués et théoriques.* PhD thesis, Université Paris 12, 2007.

[21] Attempto Project. URL: http://attempto.ifi.uzh.ch/site/description/.

[22] SAD Project. URL: http://nevidal.org/.

[23] TPTP Project. URL: www.tptp.org/.

[24] Geoff Sutcliffe. System Description: SystemOn TPTP. In *CADE*, pages 406–410, 2000.

[25] Geoff Sutcliffe. Semantic Derivation Verification: Techniques and Implementation. *International Journal on Artificial Intelligence Tools*, 15(6):1053–1070, 2006.

[26] Geoff Sutcliffe and Christian B. Suttner. The Results - of the CADE-13 ATP System Competition. *J. Autom. Reasoning*, 18(2):271–286, 1997.

[27] Geoff Sutcliffe, Christian B. Suttner, and Theodor Yemenis. The TPTP Problem Library. In *CADE*, pages 252–266, 1994.

[28] Christian B. Suttner and Geoff Sutcliffe. The Design of the CADE-13 ATP System Competition. In *CADE*, pages 146–160, 1996.

[29] TPTP Syntax. URL: www.cs.miami.edu/∼tptp/tptp/syntaxbnf.html.

[30] Jan van Eijck. Discourse Representation Theory, 2005.

[31] VeriMathDoc. URL: www.ags.uni-sb.de/∼omega/projects/verimathdoc/.

[32] Marc Wagner, Serge Autexier, and Christoph Benzmüller. PlatOmega: A Mediator between Text-Editors and Proof Assistance Systems. *Electr. Notes Theor. Comput. Sci.*, 174(2):87–107, 2007.

[33] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica.* Cambridge University Press, 1962.

[34] Claus Zinn. *Understanding Informal Mathematical Discourse.* PhD thesis, Friedrich-Alexander-Universität Erlangen Nürnberg, 2004.