Saarland University
Faculty of Natural Sciences and Technology I
Computer Science Department

Bachelor's Thesis

# Organizing a Library of Higher Order Problems

submitted by
Julian Backes

submitted
April 15, 2009

Supervisor Prof. Dr. Gert Smolka
Advisor Dr. Chad E. Brown
Reviewers Prof. Dr. Gert Smolka
Dr. Chad E. Brown

## Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, April 15, 2009

Julian Backes

## Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department.

Saarbrücken, April 15, 2009

Julian Backes

## Acknowledgement

I would like to thank Dr. Chad E. Brown for issuing this interesting topic to me, advising this work, and proofreading my thesis. I enjoyed our very interesting discussions which motivated me during the last six months.

I also thank my supervisor Prof. Dr. Gert Smolka for his great lecture *Introduction to Computational Logic* which aroused my interest for the field of logic.

Finally, I would like to thank my parents, my girlfriend, and all my friends for their continuous support.

**Abstract**

Jitpro is an interactive theorem prover for higher order logic. One specifies a problem by giving a number of sorts, constants, definitions, axioms, lemmas and some claims. Clearly, many specifications may use common basic elements such as the union of two sets.

In this thesis I will present a system for storing, retrieving and connecting these specifications using imports which finally allow us to construct new problems using already existing specifications, by instantiating sorts by types or constants by terms. Theoretical foundations about signatures, presentations and morphisms together with a proof of the Presentation Lemma will ensure that all these actions will yield consistent specifications without rendering former results (i.e. proven statements) invalid.

Using the example of Jitpro, I will show that it is very easy to integrate this system into existing theorem provers.

# Contents

# 1 Introduction

## 1.1 Jitpro

Jitpro [4] is a JavaScript Interactive Higher-Order Tableau Prover developed by Chad Brown. As the name suggests, it is used to prove higher-order claims by clicking some buttons in a web browser (without the need to install anything). In constrast to the existing automatic higher order theorem provers TPS [9] and Leo-II [10], Jitpro only assists in proving something and the user must make the main decisions himself.

Currently, the system is mainly used for research and teaching purposes. For instance, students were using it during the Introduction to Computational Logic course in 2008 at Saarland University [8] in order to prove more or less easy statements and to better understand tableau proof systems.

Jitpro is based on a set of so called *refutation rules* described in [3] which will be explained later in more detail. *Refutation* means, that, given a claim, we try to show that the negation of this claim yields a contradiction by adding more and more facts to a pool of facts (which we will call *branch* in the following) using the corresponding rules. As it is possible that the application of specific rules results in several branches, we say that we split into different branches and must refute each of these branches separately.

As a first example, consider the following rule which is one of the basic rules:

$$\textsc{Closed}\ \frac{}{A, s, \neg s \vdash \bot}$$

This rule means: Given a branch A, some term $s$ and the negation of the term $\neg s$ on A, Jitpro recognizes that this is a contradiction (which should be quite intuitive) and closes the branch. Another example is:

$$\textsc{And}\ \frac{A, s \wedge t, s, t \vdash \bot}{A, s \wedge t \vdash \bot}$$

This rule says that if the branch $A$ together with the terms $s \wedge t$, $s$ and $t$ yields a contradiction then so does $A$ together with the term $s \wedge t$. Therefore, $s$ and $t$ can be added to the branch. Jitpro does not automatically do this but it presents a button in corresponding situations which can be clicked on in order to apply the rule.

One might wonder why Jitpro is called a *tableau prover*. This is due to the fact that it does not work with proof trees (which implicitly arise from applying refutation rules) but - as already mentioned - with branches and these can be seen as a tableau. This results in the following *tableau view* of the refutations rules:

$$\textsc{Closed}\ \frac{s, \neg s}{} \qquad \textsc{And}\ \frac{s \wedge t}{s, t}$$

One of the main disadvantages of Jitpro which I will tackle in this thesis is the fact that Jitpro does not have a basic library of commonly used elements such as basic definitions of set theory or graph theory. This results in a lot of copy & paste, search & replace etc. which is not only error-prone but can also introduce logical inconsistencies which are hard to detect, if at all. It is even possible that earlier proven theorems do not hold anymore in a different context because of changes in the corresponding basic axioms (we will see an example of this in section 3.3).

## 1.2 Related Work

There has already been much research in the area of the organization of (equational) logical theories. Already in 1977, Goguen and Burstall presented in [7] the syntax of a language called *CLEAR* which was to define first order equational theories. They also described operations like *combine* or *enrich* to extend existing theories or use them as building blocks for larger theories. They even provided functions which could use theories as arguments, similar to today's ML functors as described in [11].

Fifteen years later[1], in 1992, again Burstall and Goguen presented [12] and raised their rather concrete ideas from 1977 to a completely abstract level: Instead of talking about equational theories they split this concept into *presentations* and *signatures* and first order logic was replaced by the general notion of *institutions* from which the paper derived its name. This work is very important for my thesis because most of the theoretical concepts here will rely on it and even the original *Presentation Lemma* (which we will adapt for higher order logic) comes from Goguen and Burstall.

In the same year (1992), Farmer, Guttman and Thayer argued in *Little theories* [15] why it is much better to reason in several different small theories than reasoning in one big theory. One year later, in 1993, they presented *IMPS: An Interactive Mathematical Proof System* [13] which is a proof system based on simply typed theory with partial functions and subtypes. What is so special about this system concerning this thesis is the fact that IMPS already had an *Initial Theory Library* with commonly used theories such as parts of number theory or group theory and the possiblity to reuse and *interpret* theories in order to use them in a different context. The main difference to my system will be the higher-order logic with total functions on the theoretical side and the prover independence on the practical side.

In 2000, Hutter published *Management of Change in Structured Verification* [1] where he described a truth maintenance system for existing verifications of specifications, i.e. he tried to minimize the impact of changes in specifications on corresponding verifications. These specifications were built from smaller specifications which are linked somehow resulting in a dependency graph. A similar structure will be induced by my

[1] of course, there was still a lot of important work going on in between

database because we will have several axiomatic theories linked together by *morphisms*, respectively *imports*.

The most recent (and not yet finished) work comes from Sutcliffe and Suttner, namely by their *TPTP* (Thousands of Problems for Theorem Provers) [2] which is a library of well chosen first-order problems for testing automated theorem provers. It differs from my system in the fact that the latter will be more dynamic, i.e. possibly everyone can add new and extend existing theories and problems and access single problems at any time using an online interface, whereas the former is more a fixed library of more or less independent problems which is extended from time to time by the authors.

A higher-order version of the TPTP is also available but it is still in alpha state. Nevertheless, my system will support the corresponding language called *THF0*, proposed by Benzmüller, Rabe and Sutcliffe in 2008 [14].

## 1.3 Structure of the Thesis

This thesis can roughly be split into three parts:

1. Theoretical part (chapters 2 and 3): Chapter 2 gives a short introduction to *higher-order logic* and *simply typed lambda calculus* together with the corresponding basic definitions. These will be used to introduce *signatures* and *presentations* which will, together with a corresponding *proof system*, define a *theory*.

   Chapter 3 covers the notion of *morphisms* which map between presentations. The *Presentation Lemma*, which is the central point of the theoretical part, together with its proof will tell us which preconditions have to be checked such that provability is preserved when morphing a presentation into another one.

2. Practical part (chapter 4): Chapter four will explain the most important parts of the implementation. First, the main differences to the theory developed in the two chapters before will be described. Then, we will show a datastructure which allows to very efficiently retrieve tree-like structures such as types or terms out of a relational database.

3. Outlook (chapter 5): The last chapter will give a brief outlook on how the presented ideas as well as the implementation can be further improved.

# 2 Types, Terms & Theories

Our goal in this chapter is to come up with a definition of *presentations* which allow us to specify theories such as set theory or graph theory. Presentations will be based on so called *signatures*. The idea is that a signature induces a set of terms which can be used by presentation to define more complex constructs such as axioms or claims.

## 2.1 Basic Definitions

### 2.1.1 Signatures

In simply typed higher order logic, a type is either a basic type (also called *sort*) or a functional type. Let $Sor$ be the countably infinite set of all sorts. We fix $\mathbb{B} \in Sor$ as the unique boolean sort containing only two elements 1 (*true*, $\top$) and 0 (*false*, $\bot$) and recursively define the set of all types $Typ$ using $Sor$:

$$\alpha \in Sor \cong N$$
$$\sigma, \tau \in Typ ::= \alpha \mid \sigma\,\tau$$

As we will not use all infinitely many sorts within one specification but only a certain subset of $Sor$ we are also restricted in the set of types we can use. In order to define this set, we need to know which sorts are "used" by a type. This gives us the following simple function $Sor_{Typ}$ which then allows to define the desired set:

$$\begin{aligned} Sor_{Typ} : Typ &\to 2^{Sor} \\ Sor_{Typ}\ \alpha &= \{\alpha\} \\ Sor_{Typ}\ (\sigma\,\tau) &= (Sor_{Typ}\ \sigma) \cup (Sor_{Typ}\ \tau) \end{aligned}$$

**Definition 1** (induced types)**.** Given a set of sorts $\mathcal{S} \subseteq Sor$, the set of types $\mathcal{T}(\mathcal{S})$ induced by $\mathcal{S}$ is defined as follows:

$$\mathcal{T}(\mathcal{S}) = \{\sigma \mid Sor_{Typ}(\sigma) \subseteq \mathcal{S}\}$$

Besides sorts (and the types induced by them), we also want to have so called *constants* in our specifications. But what are constants? As an example, consider the specification of the natural numbers using some sort $N$ and the Peano axioms. These say (among other things) that the *successor* of a natural number is never *zero*. The successor as well as zero are both constants because they are not concretely defined (e.g. by giving a corresponding term) but they are described by their properties (by the Peano axioms). Constants state that there is something without defining it.

| Name | Type | Meaning |
|---|---|---|
| True | 1 (or $true$ or $\top$) $\in \mathbb{B}$ | 1 (or $true$ or $\top$) |
| False | 0 (or $false$ or $\bot$) $\in \mathbb{B}$ | 0 (or $false$ or $\bot$) |
| Negation | $\neg \in \mathbb{B} \to \mathbb{B}$ | $\neg x = \lambda x.1 - x$ |
| Conjunction | $\wedge \in \mathbb{B} \to \mathbb{B} \to \mathbb{B}$ | $x \wedge y = \lambda x, y.\min\{x, y\}$ |
| Disjunction | $\vee \in \mathbb{B} \to \mathbb{B} \to \mathbb{B}$ | $x \vee y = \lambda x, y.\max\{x, y\}$ |
| Implication | $\to \in \mathbb{B} \to \mathbb{B} \to \mathbb{B}$ | $x \to y = \lambda x, y.\neg x \vee y$ |
| Equality | $=_\sigma \in \sigma \to \sigma \to \mathbb{B}$ | $(x =_\sigma y) = (\lambda x, y.x = y)$ |
| Equivalence | $\equiv \in \mathbb{B} \to \mathbb{B} \to \mathbb{B}$ | $(x \equiv y) = (\lambda x, y.x =_\mathbb{B} y)$ |
| Universal quantifier | $\forall_\sigma \in (\sigma \to \mathbb{B}) \to \mathbb{B}$ | $(\forall_\sigma.s) = (s = (\lambda x.1))$ |
| Existential quantifier | $\exists_\sigma \in (\sigma \to \mathbb{B}) \to \mathbb{B}$ | $(\exists_\sigma.s) = (s \neq (\lambda x.0))$ |

Figure 2.1: A list of all logical constants given a signature $\Sigma = (\mathcal{S}, \mathcal{C}, \tau)$ with $\sigma \in \mathcal{T}(\mathcal{S})$

At first sight, constants are just names. Let *Nam* be a countably infinite set of identifiers and let $Const \subsetneq Nam$ be the set of all names which represent constants, i.e. *Const is* the set of all constants. But as seen above, constants always have a specific type. For example *zero* will have type $N$ and the successor function will have type $N\,N$. On the other hand, it is not convenient to have a fixed mapping from constants to types: For instance, a successor function for natural numbers is different from a successor function for integers but they could both be called `Succ` in different contexts. Hence, it is better to assign types to constants depending on the context.

Sorts together with typed constants form the first important definition:

**Definition 2** (signature)**.** Given a set of sorts $\mathcal{S} \subseteq Sor$ with $\mathbb{B} \in \mathcal{S}$, a set of names $\mathcal{C} \subseteq Const$ and a function $\tau : \mathcal{C} \to \mathcal{T}(\mathcal{S})$, we call the tuple $\Sigma = (\mathcal{S}, \mathcal{C}, \tau)$ a signature.

As we will see in the next chapter, a signature is already enough to induce a set of terms which can only be extended by extending the signature.

### 2.1.2 Presentations

The main elements of a specification (e.g. axioms or lemmas) are described by terms. We want to define terms based on a signature. Let a signature $\Sigma = (\mathcal{S}, \mathcal{C}, \tau)$ be given. This signature induces a set of *logical constants* which consists of well-known logical connectives like $\wedge$ (and) or $\to$ (implication). The reason for this set to depend on a signature is that for example the forall quantifier $\forall_\sigma$ depends on a type $\sigma$ and it should hold that $\sigma \in \mathcal{T}(\mathcal{S})$. Figure 2.1.2 lists all logical constants together with their meaning.

Another (also commonly used) element of terms are lambda abstractions. They define a function taking an argument represented by a variable. Let $Var \subsetneq Nam$ such that $Var \cap Const = \emptyset$[1] be the set of all those variables.

We define the corresponding set of $\Sigma$-terms as follows:

[1] the disjointness is to avoid possible naming conflicts

$$
\begin{array}{lll}
s,t \in \Sigma\text{-}Ter ::= & \lambda x : \sigma.t & \text{Lambda expression; } \sigma \in \mathcal{T}(\mathcal{S}); x \in Var \\
& \mid s\ t & \text{Functional application} \\
& \mid x & \text{Variable; } x \in Var \\
& \mid c & \text{Constant; } c \in \mathcal{C} \\
& \mid l & \text{Logical constant}
\end{array}
$$

This definition already excludes invalid types or constants but it is still possible for a term to be ill-typed or to contain unbound variables, i.e. variables which are not bound by a corresponding lambda expresssion. In order to tackle this problem we need a function which gives us the unbound variables of a term:

$$
\begin{array}{l}
\mathcal{V} : Ter \to 2^{Var} \\
\begin{array}{rcl}
\mathcal{V}\ (\lambda x : \sigma.t) & = & \mathcal{V}\ t\ \setminus \{x\} \\
\mathcal{V}\ x & = & \{x\} \\
\mathcal{V}\ (s\ t) & = & (\mathcal{V}\ s) \cup (\mathcal{V}\ t) \\
\mathcal{V}\ c & = & \emptyset \\
\mathcal{V}\ l & = & \emptyset
\end{array}
\end{array}
$$

For a set of terms $A$, we define the free variables as:

$$\mathcal{V}\ A = \bigcup_{s \in A} \mathcal{V}\ s$$

Free variables can be *substituted* by terms. Given a *simultaneous* substitution $\theta : Var \to \Sigma\text{-}Ter$ (i.e. a substitution which substitutes multiple variables), the substitution $\overline{\theta}$ on terms is defined as follows:

$$
\begin{array}{l}
\overline{\theta} : \Sigma\text{-}Ter \to \Sigma\text{-}Ter \\
\begin{array}{rcl}
\overline{\theta}\ (\lambda x : \sigma.t) & = & \lambda y : \sigma.(\overline{\theta[x := y]}\ t) \\
\overline{\theta}\ x & = & \theta\ x \\
\overline{\theta}\ (s\ t) & = & (\overline{\theta}\ s)\ (\overline{\theta}\ t) \\
\overline{\theta}\ c & = & c \\
\overline{\theta}\ l & = & l
\end{array}
\end{array}
$$

The $y$ in the first case is chosen by a *conservative choice function* with respect to the codomain of $\theta$ which avoids variable capturing. Substitutions and choice functions are described in detail in [3]. We will use $\theta$ for substitutions on (free) variables and write $s^x_t$ to denote the substitution of free occurences of $x$ by $t$ in $s$.

**Definition 3** (closed term)**.** A term $t$ is called closed iff $\mathcal{V}\ t = \emptyset$.

Given a typing function $\upsilon : Var \to \mathcal{T}(\mathcal{S})$ on variables, we define $\overline{\upsilon}$ as a partial function on terms:

$$\overline{\upsilon} : \Sigma\text{-}Ter \rightharpoonup \mathcal{T}(\mathcal{S})$$

$$
\begin{array}{rcl}
\overline{\upsilon}\ (\lambda x : \sigma.t) & = & \sigma\ (\overline{\upsilon[x := \sigma]}\ t) \\
\overline{\upsilon}\ x & = & \upsilon\ x \\
\overline{\upsilon}\ (s\ t) & = & \sigma' \qquad \text{if } \overline{\upsilon}\ s = \sigma\ \sigma' \text{ and } \overline{\upsilon}\ t = \sigma' \\
\overline{\upsilon}\ c & = & \tau\ c \\
\overline{\upsilon}\ l & = & \text{(type according to figure 2.1.2)}
\end{array}
$$

Let $\upsilon : \mathit{Var} \to \mathcal{T}(\mathcal{S})$ be a fixed typing function. A term $s$ is well-typed for $\upsilon$ if there exists a $\upsilon$ such that $s \in \mathit{Dom}(\overline{\upsilon})$. Otherwise, $s$ is called ill-typed. Note that the well-typedness of a closed term is independent of a $\upsilon$. In such a case, we assume that some $\upsilon$ is given. We use the following notation:

- $\tau\ s$ for $\overline{\upsilon}\ s$
- $\tau\ x$ for $\upsilon\ x$

Given a signature $\Sigma = (\mathcal{S}, \mathcal{C}, \tau)$, we define $\mathit{cwff}_\sigma(\Sigma)$ as the set of all closed, well-typed $\Sigma$-terms of type $\sigma$ and $\mathit{cwff}(\Sigma)$ as the set of all closed, well-typed $\Sigma$-terms (of arbitrary type). Analogously, we use $\mathit{wff}(\Sigma)$ for all well-typed but not necessarily closed terms.

Another important primitive, which we will not need for presentations but which will be needed later, are *contexts*. Contexts are essentially terms with a hole and are defined as follows:

$$C ::= [\,] \mid s\ C \mid C\ s \mid \lambda x : \sigma.C$$

We can fill the hole of a context with a term:

$$
\begin{array}{rcl}
[\,][t] & = & t \\
(C\ s)[t] & = & C[t]\ s \\
(s\ C)[t] & = & s\ C[t] \\
(\lambda x : \sigma.C)[t] & = & \lambda x : \sigma.C[t]
\end{array}
$$

A context *captures* variables if its hole lies in the scope of a lambda operator. For example $\lambda x.[]$ captures $x$. A context $C$ is *admissible* for a term $t$ if C does not capture the free variables of $t$.

Consider a user writing a specification of set theory. Of course, he will often use the union between two sets and - given the possibilities described so far - he has to write the corresponding term each time he wants to use it. This is neither good for reuse nor for the sake of clarity. A solution to this problem are abbreviations which we will call *definitions*. These are constants in combination with terms (which of course have to match in their types), i.e. a partial function $\delta : \mathcal{C} \rightharpoonup \mathit{cwff}(\Sigma)$.

The last element of presentations - and in principle the most important one - will be *axioms* which are $\Sigma$-terms of type $\mathbb{B}$. Axioms are propositions which are assumed to be true and represent, as we will later see, the power of a presentation.

**Definition 4** (presentation)**.** Given a signature $\Sigma = (\mathcal{S}, \mathcal{C}, \tau)$, a set $\mathcal{D} \subseteq \mathcal{C}$, a function $\delta : \mathcal{D} \to \bigcup_{d \in \mathcal{D}} \mathit{cwff}_{\tau d}(\Sigma)$ such that $\tau\, d = \tau\, (\delta\, d)$ and a set $\mathcal{K} \subseteq \mathit{cwff}_{\mathbb{B}}(\Sigma)$, we call the triple $\mathcal{P} = (\Sigma, \mathcal{K}, \delta)$ a presentation.

Presentations represent specifications in theory and are used to prove claims based on a proof system induced by the respective presentation as we will see in section 2.3

But first, we will show in the next section how to write presentations in Jitpro syntax and how this relates to the theoretical definitions given so far. This will help us to give examples.

## 2.2 Presentations in Jitpro

Jitpro does not distinguish between signatures and presentations which means both are specified at the same time. For example a sort can be defined, followed by an axiom, followed by a sort again. Moreover, stating a definition implicitly defines the corresponding constant, i.e the user does not have to enter them separately (and is even not allowed to do so).

There is also a total order on the elements (which is given by the order in which the elements are typed in) so something cannot be used before it is defined. For instance, if the type of a constant contains a specific sort, this sort has to be defined before the constant is defined.

The following example defines a fragment of basic set theory:

```
1  sort I
2  const intersect: (I B) ((I B) (I B))
3  term union = \X:I B. \Y:I B. \z:I.X z | Y z
4  axiom !X:I B. !Y:I B. intersect X Y = \z:I. X z & Y z
```

This code evaluates as follows:

- The first line defines a sort $\mathbb{I}$. As $\mathbb{B}$ is defined implicitly, the set of sorts is $\{\mathbb{I}, \mathbb{B}\}$
- The second line defines a constant `intersect` of type $(\mathbb{I}\,\mathbb{B})((\mathbb{I}\,\mathbb{B})(\mathbb{I}\,\mathbb{B}))$
- The third line defines a definition, i.e. the term $\lambda X : \mathbb{I}\mathbb{B}.\ \lambda Y : \mathbb{I}\mathbb{B}.\ \lambda z : \mathbb{I}.Xz \vee Yz$ is assigned to the constant `union` of type $(\mathbb{I}\,\mathbb{B})((\mathbb{I}\,\mathbb{B})(\mathbb{I}\,\mathbb{B}))$
- The fourth line says that the intersection of two sets corresponds to a subset which contains only those elements which are in both sets. The corresponding axiom is $\forall_{\mathbb{I}\mathbb{B}}\ (\lambda X : \mathbb{I}\,\mathbb{B}.\ \forall_{\mathbb{I}\mathbb{B}}\ (\lambda Y : \mathbb{I}\,\mathbb{B}.\ \texttt{intersect}\ X\ Y =_{\mathbb{I}\mathbb{B}} \lambda z : \mathbb{I}.X\,z \wedge Y\,z))$

Altogether, we defined the signature

$$\Sigma_0 = (\{\mathbb{I}, \mathbb{B}\}, \{\texttt{union}, \texttt{intersect}\},\\ \{(\texttt{union}, (\mathbb{I}\,\mathbb{B})((\mathbb{I}\,\mathbb{B})(\mathbb{I}\,\mathbb{B}))), (\texttt{intersect}, (\mathbb{I}\,\mathbb{B})((\mathbb{I}\,\mathbb{B})(\mathbb{I}\,\mathbb{B})))\}^1)$$

and the presentation

$$\mathcal{P}_0 = (\Sigma_0, \{\forall_{\mathbb{I}\mathbb{B}}\ (\lambda X : \mathbb{I}\mathbb{B}.\ \forall_{\mathbb{I}\mathbb{B}}\ (\lambda Y : \mathbb{I}\mathbb{B}.\ \texttt{intersect}\ X\ Y =_{\mathbb{I}\mathbb{B}} \lambda z : \mathbb{I}.X\, z \wedge Y\, z))\},$$
$$\{(\texttt{union}, \lambda X : \mathbb{I}\mathbb{B}.\ \lambda Y : \mathbb{I}\mathbb{B}.\ \lambda z : \mathbb{I}.X z \vee Y z)\})$$

Jitpro supports all logical constants from figure 2.1.2 but it does not treat them as logical constants but as binary operators (`&` (conjunction), `|` (disjunction), `->` (implication), `=` (equality/equivalence), `<->` (equivalence)), unary operators (`~` (negation)) and binders (`!` (universal quanitifer), `?` (existential quantifier), \ (lambda expression)). Only `true` and `false` can be used as expected.

In particular, the type of $\forall_\sigma$, $\exists_\sigma$ and $=_\sigma$ cannot be specified. Jitpro determines it from the context in which the operators are used. This makes it impossible for Jitpro to parse the following example, even if logical operators could be used as logical constants:

```
axiom = = =
```

The type of the inner `=` depends on the type of the outer `=` which can possibly be of any type.

Besides axioms, Jitpro also supports the specification of lemmas using the keyword `lemma` instead of `axiom`. But as we will later see, they do not give us any additional power and therefore only play a role for the practical part of this thesis (see section 4).

More information about the syntax and semantics of Jitpro can be found in the online documentation at [5] and [6].

## 2.3 A Proof System for Valid Sequents

So far, we have a notion of signatures and presentations which can be used to describe axiomatic theories. The goal in this section is to come up with a notion of provability using Jitpro and presentations.

### 2.3.1 The Proof System B and Its Relation to Jitpro

Jitpro is based on a proof system called Bwhich is described in detail in [3]. B uses *logical interpretations* and the notion of *sequents* to define validity: Let a presentation $\mathcal{P} = \{\Sigma, \mathcal{K}, \delta\}$ be given. A sequent is a pair $(A, s)$ (or $A \vdash s$) such that $A \subseteq \mathit{cwff}_{\mathbb{B}}(\Sigma)$ (assumptions) and $s \in \mathit{cwff}_{\mathbb{B}}(\Sigma)$ (claim). We define the following notations for sequents:

- $A, s_0, s_1 \ldots, s_n \vdash t$ for $A \cup \{s_0, s_1, \ldots, s_n\} \vdash t$
- $s_0, s_1 \ldots, s_n \vdash t$ for $\{s_0, s_1 \ldots, s_n\} \vdash t$

A sequent $A \vdash s$ is *valid* iff whenever a logical interpretation satisfies $A$ it also satisfies $s$. Let $\models$ be the set of all valid sequents.

[1] we use sets of pairs to describe functions

**Proposition 1** (without proof)**.** *B is sound for* $\models$*.*

Proposition 1 says that if we have a closed proof tree for a sequent in B then this sequent is valid.

Possible proof steps in B are expressed by a set of inference rules. One of these rules is called CONTRA and looks as follows:

$$\textsc{Contra}\ \frac{A, \neg s \vdash \bot}{A \vdash s}$$

It can be read as "In order to prove that $A \vdash s$ is valid, we can also prove that $A, \neg s \vdash \bot$ is valid" or as "If we know that $A$ and $\neg s$ yield false then we also know that $A$ yields $s$". Using this rule, most of the basic *refutation rules* (see figures 2.2 and 2.3) used by Jitpro can be derived.

Refutation rules are used to show that something cannot hold, i.e. that it is unsatisfiable. The idea of Jitpro is that instead of proving that a sequent is valid (i.e. that a claim is true in every logical interpretation), it proves that there is no logical interpretation where the negation of the claim holds (which implies the former). In terms of sequents, Jitpro does not show that $A \vdash s$ holds but it shows that $A, \neg s$ is unsatisfiable, that is $A, \neg s \vdash \bot$, which corresponds to the CONTRA rule.

**Proposition 2** (without proof)**.** *Jitpro is sound for B.*

Proposition 2 implies that Jitpro is sound for $\models$: If there is a closed proof tree for a sequent in Jitpro there is also one in B. By proposition 1 the sequent is valid.

Proofs for propositions 2 and 1 can be found in [**?**]

### 2.3.2 The Tableau View

As already mentioned in section 1, Jitpro does not consider proofs as proof trees consisting of derivations using refutations rules (although they are applied in the background). Proof trees in Jitpro are represented as pools of facts (terms of type $\mathbb{B}$) which contain a contradiction, i.e. given any branch of a completed proof, one of the CLOSED rules was applied.

During a proof in Jitpro, the user tries to add facts to the current branch by applying refutation rules which results in a tableau view of the refutations rules. It describes, given a certain fact in a pool, which facts can be added to this pool. As an example, consider the tableau view of the rule NEGIMP:

$$\textsc{NegImp}\ \frac{\neg(s \to t)}{s, \neg t}$$

It describes the following situation: If we have $\neg(s \to t)$ in our pool (which is assumed to be satisfiable) then $s$ must be true and $t$ must be false. Therefore, $s$ and $\neg t$ can be added to the pool. Another interesting example is OR:

$$\textsc{Closed}\ \frac{}{A, \bot \vdash \bot} \qquad \textsc{Closed}\ \frac{}{A, \neg\top \vdash \bot} \qquad \textsc{Closed}\ \frac{}{A, s, \neg s \vdash \bot}$$

$$\textsc{Closed}\ \frac{}{A, \neg(s = s) \vdash \bot} \qquad \textsc{Closed}\ \frac{}{A, (s = t), \neg(t = s) \vdash \bot}$$

$$\textsc{DNeg}\ \frac{A, \neg\neg s, s \vdash \bot}{A, \neg\neg s \vdash \bot} \qquad \textsc{DeMorgan}\ \frac{A, \neg(s \vee t), \neg s \wedge \neg t \vdash \bot}{A, \neg(s \vee t) \vdash \bot}$$

$$\textsc{DeMorgan}\ \frac{A, \neg(s \wedge t), \neg s \vee \neg t \vdash \bot}{A, \neg(s \wedge t) \vdash \bot} \qquad \textsc{And}\ \frac{A, s \wedge t, s, t \vdash \bot}{A, s \wedge t \vdash \bot}$$

$$\textsc{Or}\ \frac{A, s \vee t, s \vdash \bot \qquad A, s \vee t, t \vdash \bot}{A, s \vee t \vdash \bot} \qquad \textsc{Imp}\ \frac{A, s \to t, t \vdash \bot \qquad A, s \to t, \neg s \vdash \bot}{A, s \to t \vdash \bot}$$

$$\textsc{NegImp}\ \frac{A, \neg(s \to t), s, \neg t \vdash \bot}{A, \neg(s \to t) \vdash \bot} \qquad \textsc{Lambda}\ \frac{A, s, s' \vdash \bot}{A, s \vdash \bot}\ \text{where } s \sim_\lambda s'$$

$$\textsc{Boolean=}\ \frac{A, s \equiv t, s, t \vdash \bot \qquad A, s \equiv t, \neg s, \neg t \vdash \bot}{A, s \equiv t \vdash \bot}$$

$$\textsc{Boolean}{\neq}\ \frac{A, \neg(s \equiv t), s, \neg t \vdash \bot \qquad A, \neg(s \equiv t), t, \neg s \vdash \bot}{A, \neg(s \equiv t) \vdash \bot}$$

$$\textsc{DeMorgan}\ \frac{A, \neg(\forall x.s), \exists x.\neg s \vdash \bot}{A, \neg(\forall x.s) \vdash \bot} \qquad \textsc{DeMorgan}\ \frac{A, \neg(\exists x.s), \forall x.\neg s \vdash \bot}{A, \neg(\exists x.s) \vdash \bot}$$

$$\textsc{Forall}\ \frac{A, \forall x.s, s_t^x \vdash \bot}{A, \forall x.s \vdash \bot} \qquad \textsc{Exists}\ \frac{A, \exists x.s, s_y^x \vdash \bot}{A, \exists x.s \vdash \bot}\ y \in \mathit{Var} \setminus \mathcal{V}\ (A \cup \{\exists x.s\})$$

$$\textsc{Functional=}\ \frac{A, s =_{\sigma\,\tau} u, st = ut \vdash \bot}{A, s =_{\sigma\,\tau} u \vdash \bot}\ \text{for any term t of type } \sigma$$

$$\textsc{Functional}{\neq}\ \frac{A, s \neq_{\sigma\,\tau} u, sa \neq ua \vdash \bot}{A, s \neq_{\sigma\,\tau} u \vdash \bot}\ \text{where } a \text{ is a fresh name of type } \sigma$$

$$\text{XM}\ \frac{A, s \vee \neg s \vdash \bot}{A \vdash \bot} \qquad \text{XM}\ \frac{A, \neg s \vee s \vdash \bot}{A \vdash \bot} \qquad \textsc{Weak}\ \frac{A' \vdash \bot}{A \vdash \bot}\ \text{if } A' \subseteq A$$

Figure 2.2: Basic set of refutation rules implemented by Jitpro

$$\textsc{Apply}= \frac{A, \forall \overline{x^n}.s = t, C[\overline{\theta}t], C[\overline{\theta}s] \vdash \bot}{A, \forall \overline{x^n}.s = t, C[\overline{\theta}t] \vdash \bot} \qquad \textsc{Apply}= \frac{A, \forall \overline{x^n}.s = t, C[\overline{\theta}s], C[\overline{\theta}t] \vdash \bot}{A, \forall \overline{x^n}.s = t, C[\overline{\theta}s] \vdash \bot}$$

where for both versions of the rule $\theta\ y = y$ if $y \notin \overline{x^n}$
and $C[\,]$ is admissible for $\forall \overline{x^n}.s = t$

Figure 2.3: Tableau rewriting rules implemented by Jitpro

$$\textsc{Or}\ \frac{s \vee t}{s \mid t}$$

In this case, if the original branch with $s \vee t$ is satisfiable, at least one of the extended branches is satisfiable: One, where (at least) $s$ is true and one where (at least) $t$ is true (a third branch where $s$ and $t$ are both true could also be added but this case is subsumed by the other two branches). As the idea of Jitpro is to show unsatisfiability, both branches need to be refuted.

## 2.4 Theories

We now have a proof system for propositions such as $\forall_{\mathbb{B}} x : \mathbb{B}.(x \equiv true) \vee (x \equiv false)$. So far the proof system does not depend on a presentation. We cannot yet use axioms or definitions during a proof. Let a presentation $\mathcal{P} = (\Sigma, \mathcal{K}, \delta)$ be given. We introduce two additional refutation rules depending on $\mathcal{P}$:

$$\textsc{Axiom}_{\mathcal{P}}\ \frac{A, k \vdash \bot}{A \vdash \bot}\ \text{if } k \in \mathcal{K} \qquad \textsc{ApplyDef}_{\mathcal{P}}\ \frac{A, C[c], C[\delta\ c] \vdash \bot}{A, C[c] \vdash \bot}\ \text{if } c \in Dom(\delta)$$

The first rule allows to add an axiom to a branch at any time and the second rule allows to replace a definition by the actual term. We define $\vdash_{\mathcal{P}}$ as the proof system defined by the basic refutation rules from figures 2.2 and 2.3 and the rules $\textsc{Axiom}_{\mathcal{P}}$ and $\textsc{ApplyDef}_{\mathcal{P}}$ and write $A \vdash_{\mathcal{P}} c$ if there is a closed tableau for the sequent $A, \neg c \vdash \bot$ using $\vdash_{\mathcal{P}}$.

**Definition 5** (theory)**.** Let a presentation $\mathcal{P} = (\Sigma, \mathcal{K}, \delta)$ be given

1. The $\mathcal{P}$-closure of a set $\mathcal{K}' \subseteq \mathit{cwff}_{\mathbb{B}}(\Sigma)$ is the set $\mathcal{K}'^{\bullet} = \{k \mid k \in \mathit{cwff}_{\mathbb{B}}(\Sigma) \wedge \emptyset \vdash_{\mathcal{P}'} k\}$ where $\mathcal{P}' = \{\Sigma, \mathcal{K} \cup \mathcal{K}', \delta\}$

2. $\mathcal{P}^{\bullet} = \{\Sigma, \mathcal{K}^{\bullet}, \delta\}$

3. $\mathcal{P}$ is called a $\Sigma$-theory (or just *theory* in the corresponding context) iff $\mathcal{P}^{\bullet} = \mathcal{P}$

4. $\mathcal{P}$ is said to present the theory $\mathcal{P}^{\bullet}$

Given a presentation $\mathcal{P}$, the corresponding theory $\mathcal{P}^\bullet$ are essentially all claims we can prove using $\vdash_\mathcal{P}$, i.e. all theorems of $\mathcal{P}$. At this point, it should also be clear why lemmas are not needed as a part of the definition of presentations (see section 2.2): Lemmas are (by definition) propositions which can be proven using the axioms of the corresponding presentations. This implies that lemmas are always elements of the theory of a presentation and if they are needed for a proof they can be considered as one part of the corresponding tableau which can be done "on the fly": Given a theorem $s$ the application of the XM rule adds $s \vee \neg s$ to the current branch. This new fact can be used to split into two new branches: One where $\neg s$ is added, i.e. where s can be refuted and therefore be proven and a second branch where $s$ is added and can then be used as an axiom.

## 2.5 Remarks

In this chapter, we saw how to give specifications like for instance the specification of a basic set theory. We defined *signatures* consisting of *sorts* inducing *types*, and typed *constants*. Signatures represent the components used to construct *terms*. Terms describe *axioms* and *definitions* which together form a *presentation*.

B is a proof system for valid *sequents* based on *logical interpretations*. Jitpro's *refutation rules* are derived from B using the CONTRA rule: Instead of proving that a claim holds we prove that the negation of the claim is unsatisfiable.

Proofs in Jitpro are not represented by a proof tree consisting of applications of refutation rules but by a *tableau view* which is a more intuitive representation.

We extended the basic set of refutation rules by two additional rules which depend on a presentation: AXIOM$_\mathcal{P}$ adds an axiom to the branch and APPLYDEF$_\mathcal{P}$ replaces a definition by the corresponding term. These two rules induce a presentation dependent proof system $\vdash_\mathcal{P}$. Using this proof system we defined the notion of a *theory* representing everything we can prove using a presentation.

Most of the definitions given here for higher order logic, including signatures, presentations and theories, are based on more general notions described in *Institutions* [12] by Goguen and Burstall. Note that we did not present a 1:1 translation because we used this paper more as a guideline. In the next chapter, we will introduce *morphisms* and the *Presentation Lemma* which originate from the same work.

# 3 Morphisms

Consider the following specification from section 2.2 which defines a rudimentary set theory:

```
1  sort I
2  const intersect: (I B) ((I B) (I B))
3  term union = \X:I B, Y:I B, z:I.X z | Y z
4  axiom !X:I B. !Y:I B. intersect X Y = \z:I. X z & Y z
```

If we want to define the same rudimentary set theory for subsets we would have to rewrite the specification to

```
1  sort I
2  const intersect: ((I B) B) (((I B) B) ((I B) B))
3  term union = \X:(I B) B, Y:(I B) B, z:(I B).X z | Y z
4  axiom !X:(I B) B. !Y:(I B) B. intersect X Y = \z:I B. X z & Y z
```

This transformation can be seen in two ways:

- We searched the first specification for all occurences of `I` (except for the first line) and replaced them by `(I B)`. This is a rather ad hoc view and we are not guaranteed that the changes are consistent: If we accidentally replaced `I` by `(J B)` we would have had an invalid presentation. Or we could forget to replace an `I` which yields a different presentation than the one we expected.
- We mapped the sort $I$ of the first signature to the type $(I\,B)$ of the second signature. This is a mathematical operation which is consistent because we know that $(I\,B)$ is a valid type of the second presentation. We will call this operation *morphism* (see section 3.1).

While this example looks quite trivial, consider the following specification which is to define a very basic version of graph theory:

```
1  sort V // vertices
2  const E: V (V B) // edges
3  axiom !v1:V, v2:V. (E v1 v2) -> (E v2 v1) // undirected graph
4  claim !v1:V, v2:V, v3:V. (E v1 v3) -> (union (E v1) (E v2)) v3
```

The claim is not a term of the corresponding presentation because `union` is not defined. In order to make it valid we need to insert the first specification from above after line 1, delete `sort I` and replace `I` by `V`. This is again ad hoc which is error-prone.

Alternatively, we can also define a third (bigger) presentation based on the second one and by using the first presentation. The morphism then maps from the first presentation to the third one. This operation will be called *import* (see section 3.3).

Another interesting question will be: When morphing (and importing), what happens to existing proofs? Are they still valid? Or in other words: How does a theory defined by the corresponding presentation change? Does it get bigger? Or smaller? Of course, our goal is to preserve existing proofs and this only works if we observe certain conditions. These conditions will be given by the *Presentation Lemma* in section 3.2.

## 3.1 The Definition of Morphisms

### 3.1.1 Signature Morphisms

Consider two signatures $\Sigma_1 = (\mathcal{S}_1, \mathcal{C}_1, \tau_1)$ and $\Sigma_2 = (\mathcal{S}_2, \mathcal{C}_2, \tau_2)$. A possible mapping between them could be two functions $\mu : \mathcal{S}_1 \to \mathcal{S}_2$ and $\nu : \mathcal{C}_1 \to \mathcal{C}_2$, i.e. a mapping between the sorts and a mapping between the constants. The problem with that is that it does not yield any additional power but corresponds more to some kind of renaming. As already said in the introduction to this chapter, a mapping from sorts to types and, analogously, from constants to terms is more useful. In addition, to be consistent, the type of the term constants are mapped to should correspond to the image of the type of the constant.

**Definition 6** (signature morphism)**.** Let $\Sigma = (\mathcal{S}, \mathcal{C}, \tau)$ and $\Sigma' = (\mathcal{S}', \mathcal{C}', \tau')$ be two signatures. The pair $\phi = (\mu, \nu)$ with $\mu : \mathcal{S} \to \mathcal{T}(\mathcal{S}')$ and $\nu : \mathcal{C} \to \mathit{cwff}(\Sigma')$ is called a signature morphism from $\Sigma$ to $\Sigma'$ iff $\mu\,\mathbb{B} = \mathbb{B}$ and $\nu\,c \in \mathit{cwff}_{\overline{\mu}\,(\tau\,c)}(\Sigma')$ where $\overline{\mu}$ is recursively defined on types as follows:

$$
\begin{array}{l}
\overline{\mu} : \mathcal{T}(\mathcal{S}) \to \mathcal{T}(\mathcal{S}') \\
\quad\quad \overline{\mu}\ \alpha \;=\; \mu\ \alpha \\
\overline{\mu}\ (C\ D) \;=\; (\overline{\mu}\ C)\ (\overline{\mu}\ D)
\end{array}
$$

Given a signature morphism $\phi = (\mu, \nu)$, we recursively define the function $\overline{\nu}$ on terms:

$$
\begin{array}{rcl}
\overline{\nu} : \mathit{wff}(\Sigma) \to \mathit{wff}(\Sigma') & & \\
\overline{\nu}\ (\lambda x : C.t) & = & \lambda x : (\overline{\mu}\ C).(\overline{\nu}\ t) \\
\overline{\nu}\ (s\,t) & = & (\overline{\nu}\ s)\ (\overline{\nu}\ t) \\
\overline{\nu}\ x & = & x \\
\overline{\nu}\ c & = & \nu\ c \\
\overline{\nu}\ =_\sigma & = & =_{\overline{\mu}\,\sigma} \\
\overline{\nu}\ \exists_\sigma & = & \exists_{\overline{\mu}\,\sigma} \\
\overline{\nu}\ \forall_\sigma & = & \forall_{\overline{\mu}\,\sigma} \\
\overline{\nu}\ l^1 & = & l
\end{array}
$$

[1] $l$ are in this case all logical constants except for $=_\sigma$, $\exists_\sigma$ and $\forall_\sigma$

$\overline{\overline{\nu}}$ recursively on contexts:

$$\begin{array}{rcl} \overline{\overline{\nu}}\ [\,] & = & [\,] \\ \overline{\overline{\nu}}\ (s\ C) & = & (\overline{\nu}\ s)\ (\overline{\overline{\nu}}\ C) \\ \overline{\overline{\nu}}\ (C\ s) & = & (\overline{\overline{\nu}}\ C)\ (\overline{\nu}\ s) \\ \overline{\overline{\nu}}\ (\lambda x : \sigma.C) & = & \lambda x : \overline{\mu}\,\sigma.(\overline{\overline{\nu}}\ C) \end{array}$$

and use the following notation:

- $\phi\ \alpha$ for $\mu\ \alpha$
- $\phi\ \sigma$ for $\overline{\mu}\ \sigma$
- $\phi\ c$ for $\nu\ c$
- $\phi\ t$ for $\overline{\nu}\ t$
- $\phi\ C$ for $\overline{\overline{\nu}}\ C$
- $\phi\ A$ for $\{\phi\ t \mid t \in A\}$

### 3.1.2 Theory Morphisms

Having morphisms for signatures, the next logical step would be to define morphisms on presentations. But as we will see, it makes more sense to first consider theories and talk about the preservation of provability: When mapping one theory into another one, all existing theorems should still be valid, i.e. there should exist a corresponding proof in the target theory. As already mentioned, this is very useful if, for example, a certain theorem from set theory is needed while working in a theory of natural numbers.

**Definition 7** (theory morphism)**.** Let $T = (\Sigma, \mathcal{K}, \delta)$ and $T' = (\Sigma', \mathcal{K}', \delta')$ be theories. A signature morphism $\phi$ from $\Sigma$ to $\Sigma'$ is a theory morphism from $T$ to $T'$ iff $\phi\ \mathcal{K} \subseteq \mathcal{K}'$.

This definition says that a signature morphism is also a theory morphism if and only if the morphed versions of the theorems of the source theory are also theorems of the target theory. This corresponds to the requirement from above.

On the other hand, the definition can hardly be verified for a concrete morphism: All (infinitely many) theorems of the theory would have to be checked which includes reproving the theorems to be reused. The notion of a theory morphism would not reward.

But as we will see in the next section, it is possible to reduce the number of elements which need to be checked to a finite domain by transferring the concept of theory morphisms to finite presentations.

## 3.2 The Presentation Lemma

Given a presentation $\mathcal{P} = (\Sigma, \mathcal{K}, \delta)$, consider a proof of an arbitrary theorem in the corresponding theory. The only proof steps in this proof which depend on the definitions and axioms of the presentation are proof steps which consist of the application of $\textsc{Axiom}_{\mathcal{P}}$ or $\textsc{ApplyDef}_{\mathcal{P}}$ so there are two questions:

- Are theorems elements of any theory (given some morphism) if the corresponding proof neither contains $\textsc{Axiom}_{\mathcal{P}}$ nor $\textsc{ApplyDef}_{\mathcal{P}}$?
- Does the same hold for theorems with presentation dependent proofs? And if not, are there at least certain conditions for a theorem such that it can be used within another presentation if these conditions are fulfilled?

Both questions are answered by the following lemma:

**Lemma 8** (Presentation Lemma)**.** *Let $\mathcal{P} = (\Sigma, \mathcal{K}, \delta)$ and $\mathcal{P}' = (\Sigma', \mathcal{K}', \delta')$ be presentations and $\phi$ be a signature morphism from $\Sigma$ to $\Sigma'$. $\phi$ is a theory morphism from $P^\bullet$ to $P'^\bullet$ iff $\phi\,\mathcal{K} \subseteq \mathcal{K}'^\bullet$ and $(\phi\, c =_{\phi\,(\tau\, c)} \phi\,(\delta\, c)) \in \mathcal{K}^\bullet \quad \forall c \in Dom(\delta)$*

The conditions in this lemma are only related to the definitions and axioms of a presentation. This means that, given a morphism, the morphed version of a theorem is also a theorem in the target presentation if the corresponding proof neither uses axioms ($\textsc{Axiom}_{\mathcal{P}}$) nor definitions ($\textsc{ApplyDef}_{\mathcal{P}}$) of the original presentation.

In order to prove the Presentation Lemma, two additional lemmas are needed:

**Lemma 9.** *Let $\mathcal{P} = (\Sigma, \mathcal{K}, \delta)$ be a presentation as usual and $C[t] \in wff(\Sigma)$ some context with a term in its hole. Let $\phi$ be a signature morphism from $\Sigma$ to some other signature. Then:*

$$\phi\ (C[t]) = (\phi\ C)[(\phi t)]$$

*Proof.* We prove this lemma by induction on the structure of $C$. There are four cases:

- $C = [\,]$: $\phi\ ([\,][t]) = \phi\ t = [\,]\ (\phi\ t) = (\phi\ [\,])\ (\phi\ t)$
- $C = (C'\ s')$: $\phi\ (C'\ s)[t] = \phi\ (C'[t]\ s) = (\phi\ C'[t])\ (\phi\ s) \overset{\text{IH}}{=} ((\phi\ C')[\phi\ t])\ (\phi\ s)$ $= ((\phi\ C')\ (\phi\ s))[\phi\ t] = (\phi\ (C'\ s))[\phi\ t]$
- $C = (s\ C')$: Analogously to the previous step
- $C = (\lambda x : \sigma.C')$: $\phi\ (\lambda x : \sigma.C')[t] = \phi\ (\lambda x : \sigma.C'[t]) = \lambda x : (\phi\ \sigma).\phi\ C'[t]$ $\overset{\text{IH}}{=} \lambda x : (\phi\ \sigma).(\phi\ C')[\phi\ t] = (\lambda x : (\phi\ \sigma).(\phi\ C'))[\phi\ t] = (\phi\ (\lambda x : \sigma.C'))[\phi\ t]$

□

**Lemma 10.** *Let $\mathcal{P} = (\Sigma, \mathcal{K}, \delta)$ be a presentation as usual, $s$ a well-typed $\Sigma$-Term and $\overline{\theta}$ a substitution on terms. Let $\phi$ be a signature morphism from $\Sigma$ to some other signature. Then:*

$$\phi\ (\overline{\theta}\ t) = \overline{\theta'}\ (\phi\ t)$$

*where $\theta' = \phi \circ \theta$.*

*Proof.* Proven by induction on the structure of $t$:

- $t = l$: $\phi\ (\overline{\theta}\ l) = \phi\ l = l' = \overline{\theta'}\ l' = \overline{\theta'}\ (\phi\ l)$

- $t = c$: $\phi\ (\overline{\theta}\ c) = \phi\ c = \overline{\theta'}\ (\phi\ c)$
  The last equality holds because, as above, $\phi$ cannot introduce free variables

- $t = x$: $\phi\ (\overline{\theta}\ x) = \phi\ (\theta\ x) = \theta'\ x = \overline{\theta'}\ x = \overline{\theta'}\ (\phi\ x)$

- $t = (\lambda x : \sigma.s)$: $\phi\ (\overline{\theta}\ (\lambda x : \sigma.s)) = \phi\ (\lambda y : \sigma.(\overline{\theta[x := y]}\ s))$
  $= \lambda y : (\phi\ \sigma).\phi\ (\overline{\theta[x := y]}\ s) \overset{\text{IH}}{=} \lambda y : (\phi\ \sigma).\overline{\theta'[x := y]}\ (\phi\ s)$
  $= \overline{\theta'}\ (\lambda x : (\phi\ \sigma).\phi\ s) = \overline{\theta'}\ (\phi\ (\lambda x : \sigma.s))$
  The second to last step holds because $\phi$ does not affect variables, in particular not the $y$.

- $t = (s\ t')$: $\phi\ (\overline{\theta}\ (s\ t')) = \phi\ ((\overline{\theta}\ s)\ (\overline{\theta}\ t)) = (\phi\ (\overline{\theta}\ s))\ (\phi\ (\overline{\theta}\ t))$
  $\overset{\text{IH}}{=} (\overline{\theta'}\ (\phi\ s))\ (\overline{\theta'}\ (\phi\ t)) = \overline{\theta'}\ ((\phi\ s)\ (\phi\ t)) = \overline{\theta'}\ (\phi\ (s\ t))$

□

These two lemmas finally allow to prove the Presentation Lemma:

*Proof of Lemma 8.* Assume we are given some signature morphism $\phi$, a theory $\mathcal{P}^\bullet = (\Sigma, \mathcal{K}^\bullet, \delta)$ and a closed tableau for some $k \in \mathcal{K}^\bullet$. We will show by structural induction, that there is a morphed version of this tableau which is still closed and therefore a proof for the morphed version of the problem:

- Induction base: We have to check all five CLOSED rules:
  - Given
    $$\frac{}{A, \bot \vdash \bot}$$
    consider the morphed version:
    $$\frac{}{\phi\ A, \phi\ \bot \vdash \phi\ \bot} = \frac{}{\phi\ A, \bot \vdash \bot}$$
    which is again an instance of the original CLOSED rule. The equality holds because $\phi$ leaves $\bot$ unchanged.
  - $$\frac{}{A, s, \neg s \vdash \bot}$$
    This rule is morphed to
    $$\frac{}{\phi\ A, \phi\ s, \phi(\neg s) \vdash \phi\ \bot} = \frac{}{\phi\ A, \phi\ s, \neg\ (\phi\ s) \vdash \bot}$$
    which is, as above, again an instance of the original rule.
  - The other three cases work analogously.

- Induction step: We have to check all refutation steps (which have at least one premise) whether the morphed version can still be proven. By induction hypothesis, we assume that we already have closed proof trees for all morphed premises of the step.

  –

  $$\textsc{And}\ \frac{A, s \wedge t, s, t \vdash \bot}{A, s \wedge t \vdash \bot}$$

  which is morphed to

  $$\frac{\phi\ A, \phi\ (s \wedge t), \phi\ s, \phi\ t \vdash \bot}{\phi\ A, \phi\ (s \wedge t) \vdash \bot} = \frac{\phi\ A, (\phi\ s) \wedge (\phi\ t), \phi\ s, \phi\ t \vdash \bot}{\phi\ A, (\phi\ s) \wedge (\phi\ t) \vdash \bot}$$

  This is again an instance of the And rule

  –

  $$\textsc{Imp}\ \frac{A, s \to t, t \vdash \bot \qquad A, s \to t, \neg s \vdash \bot}{A, s \to t \vdash \bot}:$$

  $$\frac{\phi\ A, \phi\ (s \to t), \phi\ t \vdash \bot \qquad \phi\ A, \phi\ (s \to t), \phi\ (\neg s) \vdash \bot}{\phi\ A, \phi\ (s \to t) \vdash \bot} =$$

  $$\frac{\phi\ A, (\phi\ s) \to (\phi\ t), \phi\ t \vdash \bot \qquad \phi\ A, (\phi\ s) \to (\phi\ t), \neg(\phi\ s) \vdash \bot}{\phi\ A, (\phi\ s) \to (\phi\ t) \vdash \bot}$$

  which is again an instance of Imp

  Most other roles work analogously, except:

  –

  $$\textsc{Lambda}\ \frac{A, s, s' \vdash \bot}{A, s \vdash \bot} \text{ where } s \sim_\lambda s'$$

  This rule is morphed to

  $$\frac{\phi\ A, \phi\ s, \phi\ s' \vdash \bot}{\phi\ A, \phi\ s \vdash \bot}$$

  This is again an instance of the Lambda rule because the lambda equivalence condition $\phi\ s \sim_\lambda \phi\ s'$ still holds:

  * $\alpha$-equivalence is still given because morphisms do not affect variable names and cannot introduce unbound variables.
  * $\beta$-reduction: Let $(\lambda x.t)\ t'$ be a $\beta$-redex in $s$ which is reduced to $t^x_{t'}$. Applying $\phi$ to $s$ yields $(\lambda x.\phi\ t)\ (\phi\ t')$ which is still a $\beta$-redex and can be reduced to $(\phi\ t)^x_{\phi\ t'} = \phi\ (t^x_{t'})$ according to Lemma 10.
  * $\eta$-reduction: Let $\lambda x.t\ x$ be an $\eta$-redex in $s$ which can be reduced to $t$. After applying $\phi$ to $s$ we get $\lambda x.(\phi\ t)\ x$ which is again an $\eta$-redex and can be reduced to $\phi\ t$.

–

$$\textsc{Apply}= \frac{A, \forall \overline{x^n}.s = t, C[\overline{\theta}t], C[\overline{\theta}s] \vdash \bot}{A, \forall \overline{x^n}.s = t, C[\overline{\theta}t] \vdash \bot}$$

Applying the morphism and Lemma 9 yields

$$\frac{\phi\ A, \forall \overline{x^n}.\phi\ s = \phi\ t, (\phi\ C)[\phi\ (\overline{\theta}t)], (\phi\ C)[\phi\ (\overline{\theta}s)] \vdash \bot}{\phi\ A, \forall \overline{x^n}.\phi\ s = \phi\ t, (\phi\ C)[\phi\ (\overline{\theta}t)] \vdash \bot}$$

According to Lemma 10, there is a $\overline{\theta'}$ such that

$$\frac{\phi\ A, \forall \overline{x^n}.\phi\ s = \phi\ t, (\phi\ C)[\overline{\theta'}\ (\phi t)], (\phi\ C)[\overline{\theta'}\ (\phi s)] \vdash \bot}{\phi\ A, \forall \overline{x^n}.\phi\ s = \phi\ t, (\phi\ C)[\overline{\theta'}\ (\phi t)] \vdash \bot}$$

which is again an instance of $\textsc{Apply}=$. The second $\textsc{Apply}=$ rule can analogously be proven.

–

$$\textsc{Axiom}_{\mathcal{P}}\ \frac{A, k \vdash \bot}{A \vdash \bot} \text{ where } k \in \mathcal{K}$$

which is morphed to

$$\frac{\phi\ A, \phi\ k \vdash \bot}{\phi\ A \vdash \bot}$$

By induction hypothesis, we have a closed tableau for $\phi\ A, \phi\ k \vdash \bot$. We give a proof tree for $\phi\ A \vdash \bot$:

$$\textsc{Weak}\ \frac{\textsc{Or}\ \dfrac{\textsc{Xm}\ \dfrac{\phi\ A, \phi\ k \vdash \bot}{\phi\ A, \phi\ k \vee \neg(\phi\ k), \phi\ k \vdash \bot} \qquad \dfrac{\neg(\phi\ k) \vdash \bot}{\phi\ A, \phi\ k \vee \neg(\phi\ k), \neg(\phi\ k) \vdash \bot}\ \textsc{Weak}}{\phi\ A, \phi\ k \vee \neg(\phi\ k) \vdash \bot}}{\phi\ A \vdash \bot}$$

The left branch can be closed by applying the induction hypothesis. The right branch can be closed because we required that $\phi\ k$ is provable, i.e. that there exists a closed tableau for $\neg(\phi\ k) \vdash \bot$.

–

$$\textsc{ApplyDef}_{\mathcal{P}}\ \frac{A, C[c], C[\delta\ c] \vdash \bot}{A, C[c] \vdash \bot} \text{ where } d \in Dom(\delta)$$

Applying the morphism and Lemma 9 yields

$$\frac{\phi\ A, (\phi\ C)[\phi\ c], (\phi\ C)[\phi\ (\delta\ c)] \vdash \bot}{\phi\ A, (\phi\ C)[\phi\ c] \vdash \bot}$$

By induction hypothesis, we have a closed tableau for $\phi\ A, (\phi\ C)[\phi\ c], (\phi\ C)[\phi\ (\delta\ c)] \vdash$

$\bot$ and want to prove $\phi\ A, (\phi\ C)[\phi\ c] \vdash \bot$. This proof is given by the following derivation:

$$\textsc{Xm}\ \frac{\textsc{Or+Weak}\ \dfrac{\textsc{Apply=+Weak}\ \dfrac{\phi\ A, (\phi\ C)[\phi\ c], (\phi\ C)[\phi\ (\delta\ c)] \vdash \bot}{\phi\ A, (\phi\ C)[\phi\ c], \phi\ c = \phi\ (\delta\ c) \vdash \bot} \qquad \neg(\phi\ c = \phi\ (\delta\ c)) \vdash \bot}{\phi\ A, (\phi\ C)[\phi\ c], \phi\ c = \phi\ (\delta\ c) \vee \neg(\phi\ c = \phi\ (\delta\ c))}}{\phi\ A, (\phi\ C)[\phi\ c] \vdash \bot}$$

As above, the left branch can be closed by applying the induction hypothesis and the right branch can be closed because we required $\phi\ c = \phi\ (\delta\ c)$ to be provable.

□

## 3.3 Imports

We now have the notion of morphisms and we know under which conditions a morphism can be used to transform a presentation into another one without losing any power. The next goal is to work towards a corresponding implementation. But an implementation of a pure morphism is not realistic.

Consider the following two examples:

1. A user is working in number theory and wants to use a theorem of set theory in some proof
2. A user is working in number theory and wants to specify sets of numbers

In the first example, the user needs to state a morphism and apply the morphism to the corresponding theorem. But before he can use this theorem, he has to prove the conditions stated in the Presentation Lemma for all axioms and definitions of set theory (or at least for those used in the original proof of the theorem).

In the second example, the user has to copy the set theory parts he wants to use, specify a morphism and apply this morphism to terms of the copied parts (e.g. terms which are used in definitions). Especially the copy part can not be handled by morphisms. To illustrate this case, recall the two code fragments from the introduction to this chapter:

```
1   sort I // set elements
2   const intersect: (I B) ((I B) (I B))
3   term union = \X:I B, Y:I B, z:I.X z | Y z
4   axiom !X:I B. !Y:I B. intersect X Y = \z:I. X z & Y z // set intersection
```

and

```
1   sort V // vertices
2   const E: V (V B) // edges
3   axiom !v1:V, v2:V. (E v1 v2) -> (E v2 v1) // undirected graph
4   claim !v1:V, v2:V, v3:V. (E v1 v3) -> (union (E v1) (E v2)) v3
```

In order to make the `union` available for graph theory, the sort $I$ needs to be morphed to $V$ and the definition of union must be copied to the second presentation while the morphism is applied to the corresponding term. If the axiom is also needed, it must also be copied and morphed. This results in the following presentation:

```
1  sort V
2  const E: V (V B)
3  const intersect: (V B) ((V B) (V B))
4  term union = \X:V B, Y:V B, z:V.X z | Y z
5  axiom !X:V B. !Y:V B. intersect X Y = \z:V. X z & Y z
6  axiom !v1:V, v2:V. (E v1 v2) -> (E v2 v1)
7  claim !v1:V, v2:V, v3:V. (E v1 v3) -> (union (E v1) (E v2)) v3
```

Copying and morphing together, or specifying a presentation and a corresponding morphism at the same time, is called *importing*:

**Definition 11** (import, single presentation)**.** Given two signatures $\Sigma_1 = (\mathcal{S}_1, \mathcal{C}_1, \tau_1)$ and $\Sigma_2 = (\mathcal{S}_2, \mathcal{C}_2, \tau_2)$ and two presentations $\mathcal{P}_1 = (\Sigma_1, \mathcal{K}_1, \delta_1)$ and $\mathcal{P}_2 = (\Sigma_2, \mathcal{K}_2, \delta_2)$, an import of $\mathcal{P}_1$ into $\mathcal{P}_2$ is an operation $\Psi$ on $\mathcal{P}_1$ and $\mathcal{P}_2$ such that $\Psi\ (\mathcal{P}_1, \mathcal{P}_2)$ yields a presentation $\mathcal{P}_3 = (\Sigma_3, \mathcal{K}_3, \delta_3)$ with $\Sigma_3 = (\mathcal{S}_3, \mathcal{C}_3, \tau_3)$ such that

- $\mathcal{S}_2 \subseteq \mathcal{S}_3$, $(\mathcal{S}_3 \setminus \mathcal{S}_2) \subseteq \mathcal{S}_1$
- $\mathcal{C}_2 \subseteq \mathcal{C}_3$, $(\mathcal{C}_3 \setminus \mathcal{C}_2) \subseteq \mathcal{C}_1$
- $\tau_3\, c = \tau_2\, c\ \forall c \in \mathcal{C}_3 \setminus C_1$, $\tau_3\, c = \phi\,(\tau_1\, c)\ \forall c \in \mathcal{C}_3 \setminus \mathcal{C}_2$
- $\mathcal{K}_2 \subseteq \mathcal{K}_3$, $(\mathcal{K}_3 \setminus \mathcal{K}_2) \subseteq \phi\,\mathcal{K}_1$
- $dom(\delta_2) \subseteq dom(\delta_3), (dom(\delta_3) \setminus dom(\delta_2)) \subseteq dom(\delta_1)$
- $\delta_3\, c = \delta_2\, c\ \forall c \in dom(\delta_3) \setminus dom(\delta_1)$, $\delta_3\, c = \phi\,(\delta_1\, c)\ \forall c \in dom(\delta_3) \setminus dom(\delta_2)$

where $\phi$ is a signature morphism from $\Sigma_1$ to $\Sigma_3$.

As it is possible to import multiple presentations, for example if a user wants to combine a theory of sets and a theory of natural numbers, we generalize the definition to the import of multiple presentations:

**Definition 12** (import, multiple presentations)**.** Let $n > 0$, let $i$ range over $\{1, \ldots, n{+}1\}$ and let $j$ range over $\{1, \ldots, n\}$. Given $n+1$ signatures $\Sigma_i = (\mathcal{S}_i, \mathcal{C}_i, \tau_i)$ and $n+1$ presentations $\mathcal{P}_i = (\Sigma_i, \mathcal{K}_i, \delta_i)$, an import of $\mathcal{P}_1 \ldots \mathcal{P}_n$ into $\mathcal{P}_{n+1}$ is an operation $\Psi$ on $\mathcal{P}_i$ such that $\Psi\ (\mathcal{P}_1, \ldots, \mathcal{P}_{n+1})$ yields a presentation $\mathcal{P}_{n+2} = (\mathcal{S}_{n+2}, \mathcal{C}_{n+2}, \tau_{n+2})$ with $\Sigma_{n+2} = (\mathcal{S}_{n+2}, \mathcal{C}_{n+2}, \tau_{n+2})$ such that

- $\mathcal{S}_{n+1} \subseteq \mathcal{S}_{n+2}$, $(\mathcal{S}_{n+2} \setminus \mathcal{S}_{n+1}) \subseteq \bigcup_j \mathcal{S}_j$
- $\mathcal{C}_{n+1} \subseteq \mathcal{C}_{n+2}$, $(\mathcal{C}_{n+2} \setminus \mathcal{C}_{n+1}) \subseteq \bigcup_j \mathcal{C}_j$

- $\tau_{n+2}\, c = \tau_{n+1}\, c\ \forall c \in \mathcal{C}_{n+2} \setminus \bigcup_j C_j$, $\tau_{n+2}\, c = \phi_j\, (\tau_j\, c)\ \forall c \in \mathcal{C}_{n+2} \cap \mathcal{C}_j$
- $\mathcal{K}_{n+1} \subseteq \mathcal{K}_{n+2}$, $(\mathcal{K}_{n+2} \setminus \mathcal{K}_{n+1}) \subseteq (\bigcup_j \phi_j\, \mathcal{K}_j)$
- $dom(\delta_{n+1}) \subseteq dom(\delta_{n+2}), (dom(\delta_{n+2}) \setminus dom(\delta_{n+1})) \subseteq \bigcup_j dom(\delta_j)$
- $\delta_{n+2}\, c = \delta_{n+1}\, c\ \forall c \in dom(\delta_{n+2}) \setminus \bigcup_j dom(\delta_j)$,
  $\delta_{n+2}\, c = \phi_j\, (\delta_j\, c)\ \forall c \in dom(\delta_{n+2}) \cap dom(\delta_j)$

where $\phi_j$ is a signature morphism from $\Sigma_j$ to $\Sigma_{n+2}$.

The example from above shows another interesting property of imports: An existing theorem from set theory can be used for a proof in the third presentation without any of the checks imposed by the Presentation Lemma. As the morphed versions of the definition and of the axiom were copied, the corresponding checks become trivial (they consist of the application of AXIOM$_{\mathcal{P}}$ and CLOSED for axioms and APPLY= and CLOSED for definitions). This makes it very easy to apply the Presentation Lemma and reuse theorems proven before.

Unfortunately, imports also inherent dangers as the following example will show. We give a specification of the natural numbers using the Peano axioms:

```
1   sort N // natural numbers
2   const 0:N // zero
3   const S:N N // successor function
4   axiom !x:N, y:N. (S x = S y) -> x = y // injectivity of S
5   axiom !x:N. S x != 0 // successor of a number is never zero
6   axiom !p:N B. p 0 & (!x:N. p x -> p (S x)) -> !x:N. p x // induction axiom
```

Assume `N` is mapped to `N B`, `0` to a set only consisting of 0, i.e. $\{0\}$[1] and `S` is mapped to a function which, given a set of numbers, returns the same set including the smallest number which is not in this set, i.e. given $\{1, 2, 4\}$, it returns $\{0, 1, 2, 4\}$ (if the set already contains all natural numbers, it is only returned). This morphism is applied to the three axioms which are imported (we use `S` as an abbreviation for the function just described):

```
1   sort N
2   axiom !x:N B, y:N B. (S x = S y) -> x = y
3   axiom !x:N B. S x != {0}
4   axiom !p:(N B) B. p {0} & (!x:N B. p x -> p (S x)) -> !x:N B. p x
```

Consider the property $\forall X : N\ B.\exists z : N.X\ z$ which says that all sets of natural numbers contain at least one number. This property is false because it does not hold for the empty set (which does not contain anything). But using the morphed induction axiom, we can prove that this property is true because it holds for the set only consisting of 0 and if it holds for a set consisting of something it still holds for a bigger set.

Being able to give an example which shows that an imported axiom is false illustrates that it is important for a user to be very careful with importing axioms.

[1] note that this is not valid Jitpro syntax but we will use it for the sake of simplicity

## 3.4 Remarks

In this chapter, we have seen how *morphisms* can be used to translate between two signatures, i.e. how to transform a term of one signature into a term of another signature.

The signature morphism can be extended to the notion of *theory morphisms*, if the morphism preserves the provability of theorems of the source theory. The problem of this definition is that its condition can hardly be verified for a concrete morphism: Infinitely many proofs must be checked.

The *Presentation Lemma* weakens these conditions and states that a signature morphism between two finite presentations is also a theory morphism between the corresponding theories if a finite number of conditions regarding the axioms and definitions of the source presentations can be checked. The proof of the Presentation Lemma is also an algorithm which translates a proof in the source presentation to a proof in the target presentation.

As pure morphisms are not powerful enough in practice, we introduced *imports* which are extended morphisms. They specify a target theory together with a corresponding morphism. Moreover, they can trivialize the checks of some conditions of the Presentation Lemma. On the other hand, when using these features, it is possible to specify inconsistent presentations which the user has to take responsibility of.

# 4 Implementation

The system is implemented using todays standard web techniques: The program code is written in object oriented PHP [16], the database holding data such as presentations is a PostgreSQL server [17]. The interface uses Javascript [24], XHTML [25] and XML [26]. This made it very easy to integrate the database into Jitpro because it uses the same techniques.

On the other hand, several problems occured during the development: Scripting languages like PHP are comparably slow and need much memory (see [20] for benchmarks). First tests showed that, without any optimization, PHP needed more than a minute to load a chain of about 300 presentation imports (i.e. a presentation which imports a presentation which imports a presentation...). To morph such a presentation, PHP's default memory limit[1] of 16 megabytes did not suffice. We needed to increase it to 160 megabytes to avoid any problems.

The first section of this chapter describes the main differences between the implementation and the theory developed in the past two chapters. In the second and in the third section, we describe how we tried to bring the two numbers from above numbers down to something reasonable and how well this worked.

## 4.1 From Theory to Practice

### Presentations and Signatures

As already described in section 2.2, Jitpro does not distinguish between signatures and presentations. They are simultaniously specified. Jitpro also has an order on the elements of a presentation, i.e. something cannot be used before it is defined. Both properties, the combination of presentations and signatures as well as the order on the elements, are inherited by our database. For example if the system checks the existence of a sort, it only considers the part which it already checked.

The same property holds for morphisms: If a morphism is applied to a sort or to a constant, the resulting type or term must be valid at this point. If a morphism is applied to a valid presentation without any errors, the resulting presentation can also be checked without any errors.

Lemmas, axioms and claims, which we will jointly refer to as *knowns* from now on, can be labeled in Jitpro. Our system even requires a unique name for each known (otherwise, the corresponding presentation will be rejected). Definitions, which do not have a corresponding constant in Jitpro, are again split into the definition part and into constant part by the system.

[1] the maximum memory which can be used by PHP to handle one call of one script

### Logical Constants

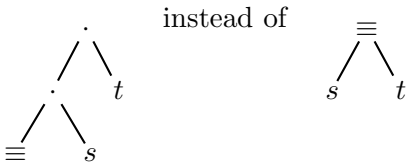
According to the definition in section 2.1.2, logical constants are a subset of all terms, i.e. $\equiv$ is already a term. In practice, and as already described in section 2.2, logical constants (except for $\top$ and $\bot$) can also be seen as operators which require certain arguments. As an example, the corresponding term for $\equiv$ would not look like

| | |
|---|---|
| (definition of terms as before) | |
| $\mid \equiv$ | equivalence |

but like

| | |
|---|---|
| (definition of terms as before) | |
| $\mid s \equiv t$ | equivalence; $s$ and $t$ are of type $\mathbb{B}$ |

The corresponding trees (section 4.2 will discuss that in more detail) look as follows:




The database is implemented using the second version because the tree representation is smaller and therefore needs less memory and administrative effort.

### Imports

Problems are caused by imports: They are handled as presentation elements, i.e. it is possible to define a sort, a constant and then an import. This makes it more complicated to do ad-hoc checks. Consider the following example:

```
1   // specification of presentation 1
2   import "presentation 2" // includes presentation 2
3   constant in:I B
```

This presentation is only valid if presentation 2 contains a sort `I` but does not contain a constant `in`. This means that, in order to check the validity of a presentation, the system needs to recursively consider imports during these checks.

Another problem is the import of big presentations. Consider the following example:

```
1   // specification of presentation 3
2   sort A, C, D, E, F, G, H, I
3   sort J, K, L, M, N, O, P, Q
4   sort R, S, T, U, V, W, X, Y
5   sort Z
```

If a user wants to import this presentation he has to tell the system what he wants to do with every single sort:

- Include the sort and apply identity morphism?
- Include the sort but with a different name and map the old sort to the new sort?
- Map the old sort to a valid type in the new presentation?

Presentations with constants, definitions and axioms make it even more complicated. To avoid such problems, we decided to implement a *default import mode* for presentation elements which is defined as follows:

- Sorts
  - If the target presentation already contains the sort, the old sort is mapped to the existing sort
  - Otherwise, the sort is imported and mapped to itself
- Constants
  - If there is already a constant with the same name in the target presentation and if the types match (i.e. the morphed type of the source constant and the type of the target constant), the old constant is mapped to the existing constant
  - If the types do not match in the first case, we exit with an error
  - Otherwise, the constant is imported and mapped to itself
- Definitions
  - If a definition with the same name already exists in the target presentation and if the terms match, we do nothing
  - If the terms do not match, we exit with an error
  - Otherwise, the definition is imported
- Knowns are handled the same way definitions are handled.

This makes it possible to write the following presentation:

```
1  // specification of presentation 4
2  sort A, C
3  import "Presentation 3"
```

The import in this presentation imports sorts `D` to `Z` and morphs `A` and `C` from presentation 3 to their counterparts in presentation 4.

Because of the implicit order of imports, the following presentation is invalid:

```
1   // specification of presentation 5
2   import "Presentation 3"
3   sort A
```

Sort `A` is imported in line 2 because it does not exist at this point. Therefore, the declaration of the same sort in line 3 is not allowed.

For knowns, the user can specify if it is imported as a claim, as a lemma, as an axiom or as it was before. There are no checks to ensure that the import is consistent. If a claim, which was never proven, is imported as a lemma, the user must be sure that it is provable using the axioms.

## 4.2 Storing Types and Terms in a Relational Database

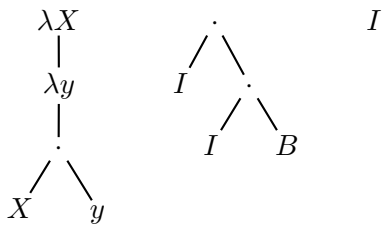
To find out where a program spends most of its execution time, it is common to use *profilers*. Profilers analyze a run of a program by measuring the time spent in functions and by counting the number of function calls. For PHP, the only good working profiler is *Xdebug* [21]. This PHP plugin has the disadvantage that it does not consider PHP's internal functions but only user defined functions which lead to disappointing results: Our functions just needed a few seconds (less than 10), i.e. they were not the reason for the long execution times.

Different tests were more promising: After disabling all database queries, the execution time reached reasonable numbers and was nearly equal to what Xdebug measured for our functions. More tests showed, that the problem did not come from single slow queries but from the total number of queries: While the time needed for a single query was very low, it turned out that we needed a few thousands queries for 300 presentations which were far too many.

Terms and types are the mostly used elements in a presentation and are therefore the first candidates to be looked at more closely. But before talking about SQL queries for loading and storing them, it makes more sense to first think about a reasonable data structure. As an example, consider the following term:

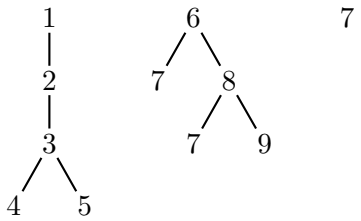
$$\lambda X : I\ (I\ B).\lambda y : I.X\ y$$

As described in [3], it is convenient to represent recursive structures as trees. The example from above can be transformed into one tree for the term and two trees for the types:



We use *size* to refer to the number of nodes in a tree. For instance the first tree has a size of 5, the last one a size of 1.

To store trees in a relational database, there are two common techniques. For their description, we abstract away from terms or types and consider arbitrary trees. If every different node in the trees from above is assigned a unique number, they look as follows:




Note, that all $I$ nodes got the same ID because they are the same node.

### 4.2.1 Pointer Representation

The first technique can be compared to an implementation of trees in a programming language using pointers or references from nodes to their children: For each node, the ID of the left child and the ID of the right child is stored. If the right child is NULL (which is used for "nothing" in SQL, comparable to a NULL pointer), the node only has one child. If both children are NULL, the node is a leaf.

Using this representation, the trees from above look as follows:

| ID | leftChild | rightChild |
|---|---|---|
| 1 | 2 | NULL |
| 2 | 3 | NULL |
| 3 | 4 | 5 |
| 4 | NULL | NULL |
| 5 | NULL | NULL |
| 6 | 7 | 8 |
| 7 | NULL | NULL |
| 8 | 7 | 9 |
| 9 | NULL | NULL |

This can be improved by using an extra table for child references which allows to have arbitrary many children:

| ID |
|---|
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |

| ID | order | childID |
|---|---|---|
| 1 | 1 | 2 |
| 2 | 1 | 3 |
| 3 | 1 | 4 |
| 3 | 2 | 5 |
| 6 | 1 | 7 |
| 6 | 2 | 8 |
| 8 | 1 | 7 |
| 8 | 2 | 9 |

The order on the children is important because there is an order on the children in terms and types. The pointer representation has the following properties:

- $\mathcal{O}(size)$ queries are needed for storing a tree (at least, each node must be stored)
- $\mathcal{O}(size)$ queries are needed for loading a tree (the nodes must recursively be queried)
- It is cheap and easy to edit a tree: Only the leftChild and rightChild columns respectively the corresponding rows in the second table must be changed.
- Some queries can be saved by caching already loaded nodes. For instance the node 7 in the second tree only needs to be loaded once
- Similarly, some space can be saved because each unique node only needs to be stored once

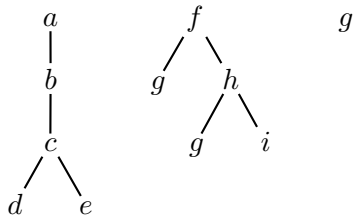
As presentations are loaded much more often than stored (and never edited), it is important to have a low number of queries for loading a presentation. Using the pointer structure, loading the term from above already takes at least 9 queries using the simple version and 17 queries using the more complex (but also more flexible) version. For bigger terms using bigger types, 80 queries would not be unusual. Although this does not sound very practical, the pointer representation has been the preferred structure in the past, for example in [22] and [23].

Nevertheless, the optimal case would be to have the flexibility of the second version while having less queries than we would have when using the simple version. This can be realized by the *nested set* representation.

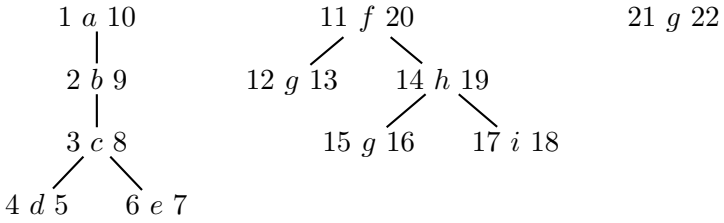
### 4.2.2 Nested Set Representation

The theory behind relational databases is set theory, i.e. relational databases should have the ability to execute set-related operations very fast. Based on this assumption, Kamfonas described in [19] a tree representation working with sets. Given two of these sets, they are either disjoint or one is a subset of the other. The subset relation lead to the name *nested sets*, given by Celko in [18].

The connection to trees comes from the fact that a tree is either a subtree of another tree (subset) or completely different from it, e.g. siblings or independent trees (disjoint sets). Recall the example from above, using letters instead of numbers:




Running a depth first search on these trees while recording the discover and the finish times, yields the following trees:




These numbers have a few nice properties considering an arbitrary node, for example (we use $l$ ro refer to the left number and $r$ to refer to the right number):

- $l$ is smaller and $r$ is bigger than any number in any subtree of the node.
- $\lfloor \frac{r-l}{2} \rfloor$ corresponds to the number of children of a node. If this number is 0, the node is a leaf.
- The path from the root to the node are all nodes with a smaller $l$ and a bigger $r$.

We are especially interested in the first property: To get a whole tree corresponding to some root, we just need to select all nodes which have a bigger $l$ and a smaller $r$ than the corresponding values of the root. This can be done in one single query.

On the other hand, it is not possible to save space by using redundancies. This is illustrated by the corresponding database table:

| ID | letter | left | right |
|---|---|---|---|
| 1 | a | 1 | 10 |
| 2 | b | 2 | 9 |
| 3 | c | 3 | 8 |
| 4 | d | 4 | 5 |
| 5 | e | 6 | 7 |
| 6 | f | 11 | 20 |
| 7 | g | 12 | 13 |
| 8 | h | 14 | 19 |
| 9 | g | 15 | 16 |
| 10 | i | 17 | 18 |
| 11 | g | 21 | 22 |

As the structure of a tree is determined by the left and right values, there is no possibility to reuse existing trees as parts of new trees (a new tree will have completely different left and right values). Only if a new tree exactly matches an existing (sub-)tree, the new tree does not need to be stored at all. In our example, the last row would not need to be stored because it can be represented by row 7 or by row 9.

To sum up, the nested set representation has the following properties:

- $\mathcal{O}(size)$ queries needed for storing a tree
- $\mathcal{O}(1)$ queries needed for loading a tree
- Editing a tree requires complex operations, but it can be done in $\mathcal{O}(1)$ queries
- Space and queries can only be saved in certain special cases

### 4.2.3 Comparing both structures in practice

For testing the performance of both structures under PHP, we created a full binary tree with 2047 nodes but only with three different leafes, i.e. we expected a lot of redundancies. Indeed, the recursive structure only needed 343 rows whereas the table of the nested set structure had (as expected) 2047 rows.

The test system was a Fedora Linux in a XEN virtual machine running on an AMD Athlon 64 X2 5600+ Dual Core with 2 GB DDR2 RAM and a 400GB SATAII hard disk.

Figure 4.2.3 shows the results of the benchmark: The nested set structure is more than twice as fast as the recursive structure (0.12 seconds compared to 0.28 seconds). If there were less redundancies, the difference would be even bigger because the recursive structure would need even more queries.

### 4.2.4 Finding Existing Terms and Types

As mentioned in the last section, the nested set structure only allows to save space if a new tree matches an existing (sub-)tree. But how can we find out if two trees match? SQL does not allow to match multiple rows against each other.

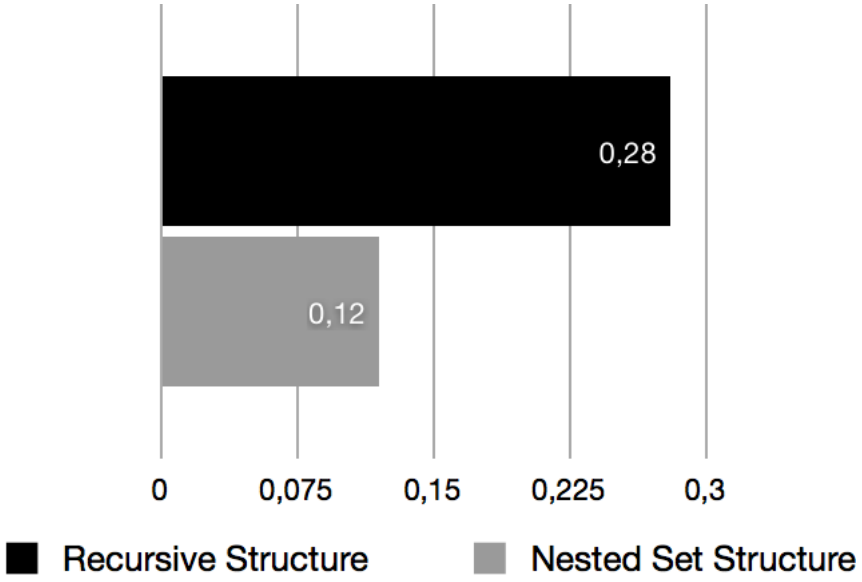



Figure 4.1: Seconds needed for loading a full binary tree with 2047 nodes

To solve this problem, we store for each row a text representation of the tree represented by this row. This text representation is a mix between Jitpro syntax and the XML code used for communication between Jitpro and the database (see Appendix A). As an example, consider the text representation from the example from the last section:

```
\X:ar(s(I),ar(s(I),b()).\y:s(I).(v(X) v(y))
```

To avoid invalid representations, the system must ensure that user defined names do not contain brackets or other characters like spaces or commas.

With this additional information, it is easy to check whether a specific type or term already exists: The database only needs to compute the text representation and search the database for it. This is very fast due to smart indices and algorithms provided by today's database systems.

## 4.3 Union Queries

Another place which needed many queries was the loading of a whole presentation. The corresponding function worked as follows:

1. Load presentation data (name, description...)
2. Load sorts of the presentation
3. Load constants of the presentation

4. Load definitions of the presentation
5. Load knowns of the presentation
6. Load imports of the presentation

Each of these steps took one database query, i.e. 300 presentations needed 1800 queries just for this one function. The separated queries even caused an additional problem concerning the order on the presentation elements. Consider the following two tables for sorts and constants (`orderID` states the order within the corresponding presentation):

| sortID | presentationID | orderID | sortName |
|---|---|---|---|
| 1 | 1 | 1 | `I` |
| 2 | 1 | 3 | `N` |

| constantID | presentationID | orderID | constantName | typeID |
|---|---|---|---|---|
| 1 | 1 | 2 | `element` | 1 |
| 2 | 1 | 4 | `one` | 2 |

which could correspond to the presentation (depending on the type of the constants):

```
1  sort I
2  const element:I
3  sort N
4  const one:N
```

As we load the data in the two tables separately, the SQL database can order them only separately. To restore the whole presentation, we need to merge the two results which is again relatively slow due to PHP. A better solution would be to combine all the results already at the database level and order them as one big table.

SQL provides so-called *union queries* which combine the result sets of multiple select statements and even allow to order this bigger set according to a specific column. At first sight, this exactly corresponds to the requirements from above. However, result sets can only be combined if they have the same table structure. For example the two tables from above have clearly different structures (the number of columns does not even match). But even with the same table structure, it would be hard to distinguish the results from one table from the results of another table: Imagine two separate tables for axioms and lemmas. They would both have exactly the same structure so how can they be distinguished?

In SQL, it is possible to add new colums on the fly. For example the query

```
1  SELECT
2      'bla' AS "column1",
3      'blub' AS "column2";
```

implicitly returns a table consisting of two columns named `column1` and `column2` containing `bla` respectively `blub`. This technique cannot only be used to add a distinguishing column but can also be used to make tables structurally equal. For the example tables from above, this would look as follows:

```
1   (
2       SELECT
3           'sort' AS "rowType",
4           "sortID" AS "id",
5           "presentationID",
6           "sortName" AS "name",
7           "orderID",
8           0 AS "typeID"
9       FROM
10          sorts
11  ) UNION (
12      SELECT
13          'constant' AS "rowType",
14          "constantID" AS "id",
15          "presentationID",
16          "constantName" AS "name",
17          "orderID",
18          "typeID"
19      FROM
20          constants
21  ) ORDER BY "orderID"
```

This query merges as many columns as possible. For instance `sortName` and `constantName` are merged to `name`. As `typeID` is is the only column which cannot be merged, it must be added to the result of `sorts`. In this case, we use `0` as the corresponding default value such that the database can match the types.

The column `rowType` can be used to distinguish rows of the two tables: If it contains `sort` it belongs to the `sorts` table, if it contains `constant`, it belongs to `constants`.

For presentations, the use of union queries reduces the minimal number of queries from 6 to 2 which results in 600 queries instead of 1800 queries for 300 presentations. Using the same technique for imports (which have a similar structure like presentations) reduces the number of queries even further.

## 4.4 Reducing Memory Consumption

After the implementation of the nested set structure and of the union queries, the time needed to load a chain of 300 presentation imports decreased (as expected) by more than a half to about 10 seconds. But the memory consumption as described in the introduction to this chapter was still a problem.

It is clear that, in order to load a presentation, each distinct presentation element (sorts, constants, types, terms...) needs to be loaded (and therefore stored in memory) at least once. Caching and referencing those elements avoids loading the same thing multiple times.

But what happens when morphing a presentation? Changing referenced elements is a bad idea because this does not only change the one occurence which we wanted to morph but changes all referenced occurences. At first sight, copying the presentation (and therefore of all corresponding cached elements) seems to be a practical solution. On the other hand, this (at least) doubles the memory consumption and causes the problems already described.

Our solution is a mix between copying and caching: In practice, a user does normally not morph presentations but only includes them, i.e. implicitly applies the identity morphism. In these cases, most terms and types do not change at all and can be reused without copying them. To check whether a term or a type changes, an algorithm only needs to check if they contain constants or sorts which are not morphed to themselves. If they contain such elements, the term or type is copied and morphed and otherwise just reused.

Using this technique, we were able to reduce the memory consumption for morphing 300 presentations from 160 MB to 128 MB.

## 4.5 Remarks

Our system is implemented using PHP and PostgreSQL. In contrast to the definitions in chapters 2 and 3, we combine signatures and presentations and speek only about presentations.

Logical constants are handled as logical operators because the corresponding tree representations are smaller. Imports have a default import mode for presentation elements to save users from stating for each single presentation element what to do with it.

As PHP is comparably slow and has a high memory consumption, a chain of 300 presentation imports already needs over one minute to load and allocates about 160 MB memory when morphed.

Using a profiler, we found out that most time is spent in thousands of database queries. To reduce this number, we implemented terms and types using the nested set structure which supports the loading of trees in one query.

Union queries reduce the number of queries even further: Instead of loading presentation elements like sorts and constants separately, they are retrieved in only one big query which combines multiple queries by using the `UNION` statement. In addition, this makes it possible to sort the presentation elements on the database level which is much faster than using PHP.

The high memory consumption when morphing is reduced by copying only those elements which are really affected by the morphism. If a term or type does not change it is simply reused.

# 5 Conclusions

In this thesis, we have shown that the ideas of Goguen and Burstall are applicable to simply typed higher order logic using Jitpro as the corresponding proof system: Signatures and presentations together with Jitpro define the notion of a theory. Morphisms translate between different theories. To ensure that such a translation does not render any former proof invalid, the conditions stated by the Presenation Lemma need to be observed. We showed that this lemma indeed holds for simply typed higher order logic.

The implementation of a problem and theory database in PHP is tricky: PHP is comparably slow and has a high memory consumption. One main goal was to reduce the number of database queries: We presented two structures, the nested set structure and the pointer structure, which can be used to store trees in a relational database. Trees are important because recursive structures such as terms or types can easily be represented by them.

The nested set structure turned out to be very efficient when loading trees: Only a constant number of queries is needed whereas the pointer structure needs a linear number of queries depending on the size of the tree. As presentations are loaded much more often than stored, we decided to use the nested set structure.

To reduce the high memory consumption when morphing a presentation into another, we only copy presentation elements which are really affected by the morphism. This technique resulted in a nearly 20% lower memory usage.

## 5.1 Future Work

There are several ways to improve or extend our system:

- In order to avoid the problems described in section 3.3, morphisms could be extended to carry predicates with them specifying subsets of types. In the corresponding example, we tried to morph the sort of the natural numbers `N` to subsets of natural numbers `N B`. This resulted in an inconsistent presentation because one of the axioms did not hold anymore. The counterexample used the fact that the type `N B` also includes the empty subset.

  To avoid that, we could also define a predicate which says that nothing of type `N B` equals the empty set and apply the morphism with repect to that predicate.

- Currently, non-default imports and morphisms can only be specified using XML code. It would be more convenient if there was corresponding Jitpro syntax like we used it in section 4.1, for example

```
1  import "Presentation xyz"
2      rename sort I = J // imports sort I as sort J
3      morph sort C = C B // morphs sort C to type C B
4  end import
```

As a user does possibly not know which elements are contained in a specific presentation, another possibility would be to present one text field for each element in the presentation to be imported. On the other hand, depending on the size of this presentation, there can be too many text fields.

- It is possible that the database will get very big depending on its usage, i.e. that there are many specifications and problems from many different theories. Currently, a user has no possibility to search the database for existing presentation elements, such as specific definitions or axioms.

  While it is easy to search for their names, it is much more difficult to search for partial terms or partial types because higher order unification is undecidable. Nevertheless, it is possible to retrieve certain candidates by using special indices as described for example in [22] or [23].

# A XML DTD for the communication between Jitpro and the database

```
<!ENTITY % PRESENTATIONITEM "sort | def | const | axiom
 | lemma | claim | import">
<!ENTITY % TYPE "B | s | ar">
<!ENTITY % TERM "v | c | d | true | false | not
 | and | or | imp | equiv | ap | l | all | ex | exu | eq">
<!ENTITY % IMPORTITEM "sortsubst | sortimport | constsubst
 | constimport | defimport | knownimport">

<!ELEMENT presentations (presentation*)>

<!ELEMENT presentation (%PRESENTATIONITEM;)*>
<!ATTLIST presentation
    name CDATA #REQUIRED
>

<!-- PRESENTATIONITEM -->
<!ELEMENT sort EMPTY>
<!ATTLIST sort
    name CDATA #REQUIRED
>
<!ELEMENT def ((%TYPE;), (%TERM;))>
<!ATTLIST def
    name CDATA #REQUIRED
>
<!ELEMENT const (%TYPE;)>
<!ATTLIST const
    name CDATA #REQUIRED
>
<!ELEMENT axiom (%TERM;)>
<!ATTLIST axiom
    name CDATA #IMPLIED
>
<!ELEMENT lemma (%TERM;)>
<!ATTLIST lemma
    name CDATA #IMPLIED
```

```
>
<!ELEMENT claim (%TERM;)>
<!ATTLIST claim
    name CDATA #IMPLIED
>

<!ELEMENT import (%IMPORTITEM;)*>
<!ATTLIST import
    dompresentationname CDATA #REQUIRED
>

<!-- IMPORTITEM -->
<!ELEMENT sortsubst (%TYPE;)>
<!ATTLIST sortsubst
    name CDATA #REQUIRED
>

<!ELEMENT sortimport EMPTY>
<!ATTLIST sortimport
    name CDATA #REQUIRED
    newname CDATA #IMPLIED
>

<!ELEMENT constsubst (%TERM;)>
<!ATTLIST constsubst
    name CDATA #IMPLIED
>

<!ELEMENT constimport EMPTY>
<!ATTLIST constimport
    name CDATA #REQUIRED
    newname CDATA #IMPLIED
>

<!ELEMENT defimport EMPTY>
<!ATTLIST defimport
    name CDATA #REQUIRED
    newname CDATA #IMPLIED
>

<!ELEMENT knownimport EMPTY>
<!ATTLIST knownimport
    name CDATA #REQUIRED
    newname CDATA #IMPLIED
```

```
    importas (lemma|axiom|claim|untouched) "untouched"
>

<!-- TYPE -->
<!ELEMENT B EMPTY>
<!ELEMENT s EMPTY>
<!ATTLIST s
    name CDATA #REQUIRED
>
<!ELEMENT ar ((%TYPE;), (%TYPE;))>

<!-- TERM -->
<!ELEMENT v EMPTY>
<!ATTLIST v
    name CDATA #REQUIRED
>
<!ELEMENT c EMPTY>
<!ATTLIST c
    name CDATA #REQUIRED
>
<!ELEMENT d EMPTY>
<!ATTLIST d
    name CDATA #REQUIRED
>
<!ELEMENT true EMPTY>
<!ELEMENT false EMPTY>
<!ELEMENT not (%TERM;)>
<!ELEMENT and ((%TERM;),(%TERM;))>
<!ELEMENT or ((%TERM;),(%TERM;))>
<!ELEMENT imp ((%TERM;),(%TERM;))>
<!ELEMENT equiv ((%TERM;),(%TERM;))>
<!ELEMENT ap ((%TERM;),(%TERM;))>
<!ELEMENT l ((%TYPE;),(%TERM;))>
<!ATTLIST l
    varname CDATA #REQUIRED
>
<!ELEMENT all ((%TYPE;),(%TERM;))>
<!ATTLIST all
    varname CDATA #REQUIRED
>
<!ELEMENT ex ((%TYPE;),(%TERM;))>
<!ATTLIST ex
    varname CDATA #REQUIRED
>
```

```
<!ELEMENT exu ((%TYPE;),(%TERM;))>
<!ATTLIST exu
    varname CDATA #REQUIRED
>
<!ELEMENT eq ((%TYPE;),(%TERM;),(%TERM;))>
```

# Bibliography


[1] Dieter Hutter: *Management of change in structured verification.* In Proceedings of the 15th IEEE International Conference on Automated Software Engineering, number 2000 in ASE. 23–34, 2000.

[2] Geoff Sutcliff, Christian Suttner, Theodor Yemenis: *The TPTP Problem Library.* In Proceedings of the 12th International Conference on Automated Deduction, 252–266, 1994
`http://www.tptp.org`

[3] Gert Smolka, Chad E. Brown: *Introduction to Computational Logic - Lecture Notes SS 2008.* 2008.

[4] Chad E. Brown: *Jitpro, A JavaScript Interactive Higher-Order Tableau Prover.*
`http://www.ps.uni-sb.de/jitpro/`

[5] Chad E. Brown: *Jitpro - Grammar for the prover*
`http://www.ps.uni-sb.de/jitpro/grammar.html`

[6] Chad E. Brown: *Jitpro - Help for the Prover*
`http://www.ps.uni-sb.de/jitpro/help.html`

[7] R.M. Burstall, J.A. Goguen: *Putting theories together to make specifications.* In Proceedings of the 5th International Joint Conference on Artificial Intelligence, 1045–1058, 1977.

[8] Gert Smolka: *Introduction to Computational Logic - Course Webpage*
`http://www.ps.uni-sb.de/courses/cl-ss08/`

[9] Peter B. Andrews, Chad E. Brown: *TPS: A hybrid automatic-interactive system for developing proofs.* Journal of Applied Logic, Volume 4, 367–395, 2006.

[10] Christoph Benzmüller, Lawrence C. Paulson, Frank Theiss, Arnaud Fietzke: *LEO-II - A Cooperative Automatic Theorem Prover for Classical Higher-Order Logic (System Description).* In Proceedings of the 4th international joint conference on Automated Reasoning, 162–170, 2008.

[11] Robin Milner, Mads Tofte, Robert Harper, David MacQueen: *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[12] J.A. Goguen, R.M. Burstall: *Institutions: Abstract Model Theory for Specification and Programming.* Journal of the ACM, Volume 39, 95–146, 1992.

[13] William M. Farmer, Joshua D. Guttman, F. Javier Thayer: *IMPS: an Interactive Mathematical Proof System*. Journal of Automated Reasoning, Volume 11, 653–654, 1993.

[14] C. Benzmüller, F. Rabe, G. Sutcliffe: *THF0 - The core of the TPTP Language for Higher-Order Logic*. In Proceedings of the 4th international joint conference on Automated Reasoning. 491-506, 2008.

[15] William M. Farmer, Joshua D. Guttman, F. Javier Thayer: *Little Theories*. In Proceedings of the 11th International Conference on Automated Deduction, 567–581, 1992.

[16] *PHP: Hypertext Preprocessor*
`http://www.php.net`

[17] *PostgreSQL: The world's most advanced open source database*
`http://www.postgresql.org/`

[18] Joe Celko: *Joe Celko's SQL for Smarties*. Morgan Kaufmann, 2005.

[19] Michael J. Kamfonas: *Recursive Hierarchies: The Relational Taboo!*. The Relational Journal, 1992.
`http://kamfonas.com/id3.html`

[20] *The Computer Language Benchmark Game*
`http://shootout.alioth.debian.org/`

[21] *Xdebug - Debugger and Profiler Tool for PHP*
`http://www.xdebug.org`

[22] Peter Gursky: *Storage and Retrieval of First Order Logic Terms in a Database*

[23] Paul Singleton, Pearl Brereton: *Storage and Retrieval of First-order Terms using a Relational Database*. In Proceedings of the 11th British National Conference on Databases: Advances in Databases. 199-219, 1993.

[24] Ecma International: *Standard ECMA-262: ECMAScript Language Specification*. 3rd edition, 1999.
`http://www.ecma-international.org/publications/standards/Ecma-262.htm`

[25] World Wide Web Consortium: *XHTML 1.0 The Extensible HyperText Markup Language (Second Edition)*. 2002.
`http://www.w3.org/TR/xhtml1/`

[26] World Wide Web Consortium: *Extensible Markup Language (XML) 1.1 (Second Edition)*. 2006.
`http://www.w3.org/TR/xml11/`