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

Bachelor's thesis

Translating a Satallax Refutation to a Tableau Refutation Encoded in Coq

submitted by

Andreas Teucke

submitted

April 29, 2011

Supervisor

Prof. Dr. Gert Smolka

Advisor

Dr. Chad E. Brown

Reviewers

Prof. Dr. Gert Smolka

Dr. Chad E. Brown

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,

Date	Signature

Abstract

Satallax is an automated theorem prover for higher-order logic. It works by reducing theorem proving to a sequence of SAT problems, which are checked by Minisat for satisfiability. If Satallax is successful in finding a proof, the result is an unsatisfiable set of clauses and a table assigning higher-order formulas to the propositional literals.

In this thesis we will present an algorithm that takes the output of Satallax and generates a tableau refutation as a proof script for the proof management system Coq to check. The algorithm requires just an UNSAT-core from the set of clauses and uses them to generate a finite tableau calculus in which we will search for a refutation. A formal proof of a refutation's existence in such a calculus will ensure the success of our algorithm.

Acknowledgment

My deepest gratitude is due to Dr. Chad E. Brown for offering me such an interesting topic. Our meetings were very interesting and I was looking forward to them each week. His explanations were invaluable for my understanding of the subject matter, and his advice often showed me the right direction when I was too focused on one part.

I also thank my supervisor and mentor Prof. Dr. Gert Smolka for his great lecture Introduction to Computational Logic, for his advice throughout my undergraduate studies and for reviewing my thesis.

Finally, I am deeply grateful to my family for their support and reminding to work when I was too lazy and to take breaks when I was working too much.

Contents

1 Introduction

In this thesis we present an extension to the automated higher-order theorem prover Satallax [2]. Satallax works by reducing higher-order proving to a sequence of propositional satisfiability problems [10]. The backbone of the procedure is a SAT solver that solves these SAT problems efficiently. If Satallax succeeds we will get as an answer just yes or no, but no proof. There are several reasons why someone might distrust a such result:

Is the method Satallax uses correct? Is it correctly implemented or are there bugs in the implementation? Even if it is correct, the SAT solver could be incorrect. The implementation is intended to allow to change the solver, which might be too complex to ensure its correctness. Furthermore, many solvers will not produce a proof, if the problem is unsatisfiable. Even though some do, does this justify the result for the higher-order problem?

Such trust issues have existed since computers are used to solve mathematical problems. De Bruijn already described this issue when he introduced AUTOMATH [13]:

> There is a case for automatic proof writing in AUTOMATH if we have to produce a tedious long proof along lines that can be precisely described beforehand. Let us take an example. Assume that P is a proposition on magic squares, and that we want to prove a theorem saying that there is no 8x8 magic square that has property P. We can write a computer program for this and run it on a computer. The computer says that none exist. Now quite apart from the question whether the computer is right, we have to admit that a formal mathematical proof has not been produced. Even if we had a complete mathematical theory about the machine, the machine language, the programming language, our proof would depend on intuitive feelings that the program gives us what we want, and it would definitely depend on a particular piece of hardware. For those who are willing to take AUTOMATH, at least temporarily, as their only final conscience of mathematical rigour, there is a way out.

> We can rewrite the magic square program in such a way that the search is stepwise accompanied by the production of AUTOMATH lines that give account of a detailed mathematical reasoning, ending with the conclusion that there is no 8x8 magic square with property P. This way we get a complete proof that can be checked by any mathematician. If we leave the checking to a computer, again we get into the question of whether the processor and the computer do what we expect them to do, but that is an entirely different matter. [13, Section 11.2]

We want to follow his suggestion for this thesis, but instead of AUTOMATH we will use the modern system COQ [1]. COQ is widely trusted and a result given as a proof script can be verified independently from Satallax. However, we want to avoid rewriting Satallax such that it is "stepwise accompanied by the production of [COQ] lines", as this would include rewriting the SAT solver. Furthermore, Satallax often produces SAT problems with thousands of clauses, while only a few are necessary. Of course, there is no way for Satallax to know them in advance.

Therefore we produce our proof in a post-processing phase after the main search of Satallax has finished. As the search algorithm is based on a cut-free tableau calculus for Church's simple type theory with choice [6], the natural choice for our proof is to construct a tableau refutation. Additionally, we will use the result of Satallax to restrict the tableau calculus for each individual refutation. We first prove that the restricted calculus does contain a refutation by giving a constructive proof and then we describe our implementation.

1.1 Structure of the Thesis

This thesis is split into five parts:

1. Preliminaries (Chapter 2): The first section of Chapter 2 gives basic definitions for higher-order logic and tableau calculus as well as introduces the underlying programs - SATALLAX, MINISAT and COQ. Then we will formulate the conjecture the thesis is based on. In the last section we give two examples that demonstrate the conjecture and disclose the first issues we have to solve.

2. Theoretical part (Chapter 3): In Chapter 3 we will formalize and prove our conjecture. We begin with descriptions of the necessary definitions and then give the prove in two parts - a lemma and the actual theorem.

3. Practical part (Chapter 4): Chapter 4 will describe the implementation. It is itself again split into three parts. In the first part we give the algorithm that constructs a refutation "skeleton" and some details about the implementation. In the second part we describe what the skeleton is missing and how to fill those holes efficiently. In the last part we show how the final refutation is simulated in Coq.

4. Results part (Chapter 5): Chapter 5 shows some results of problems from the TPTP library.

5. Final part (Chapters 6): In the last chapter we will give an outlook how the implementation can be further improved and what cases the implementation does not cover.

2 Fundamentals

In this chapter we give the notations used in this thesis and some information about the underlying software we use. The notations are mostly consistent with [6] and [10].

2.1 Preliminaries

We first present the syntax of Church's simple type theory with a choice operator. A more thorough description can be found in [6]. Types, σ, τ, μ, are inductively defined by the base types o, ι and $\sigma \to \tau$ - truth values, individuals and functions from σ to τ respectively. Function types $\sigma \to \tau$ are abbreviated by $\sigma\tau$ with right-associativity for parenthesis, e.g., $\sigma\tau\mu$ is equal to $\sigma(\tau\mu)$. There is a countably infinite set of variables for each type σ. Each term is typed and a term of type o is called a *formula*.

We give the logical constants and their types in Figure 2.1. $\bot$, $\to$, $\forall_\sigma$, $=_\sigma$, ε_σ are called *standard* constants and a formula is *standard* if it only contains standard constants. Because we are in a classical, extensional setting, there is a clear process by which a formula s can be transformed into a standard formula s' such that s and s' are equivalent. We refer to this process as standardisation. We will return to this in Section 4.3.1. We use common notations for those constants: Infix notation for $\to$, $=_\sigma$, $\vee$, $\wedge$, $\leftrightarrow$ and binder notation for $\forall_\sigma$, $\exists_\sigma$ and ε_σ, e.g., $s = t$ and $\forall x.s$ instead of $=_\sigma s\ t$ and $\forall_\sigma \lambda x.s$. We will also use $\neg$ as a notation for $\to \bot$ in standard formulae. The set $\mathcal{V}t$ of *free variables of t* has the usual definition.

The normalization of a term s is denoted by $[s]$ and s is *normal* if $[s] = s$. For the theoretical part of the thesis we will assume $[.]$ as $\beta\eta$-normalization. The implementation, however, uses different kinds of "normalization" which will be explained in Section 4.3.1.

logical constant	type	description
$\bot$	o	false
$\rightarrow$	ooo	implication
$\forall_\sigma$	$(\sigma o)o$	forall quantification
$=_\sigma$	$\sigma\sigma o$	equality
ε_σ	$(\sigma o)\sigma$	choice operator
$\top$	o	true
$\neg$	oo	negation
$\vee$	ooo	disjunction
$\wedge$	ooo	conjunction
$\leftrightarrow$	ooo	equivalence
$\neq_\sigma$	$\sigma\sigma o$	inequality
$\exists_\sigma$	$(\sigma o)o$	exist quantification

Figure 2.1: List of logical constants. The first five are *standard* constants. The remaining constants can be expressed in terms of the first five.

2.1.1 Tableau Calculus

A *branch* is a finite set of normal higher-order formulae. A *standard* branch contains only standard formulae. A branch A is *closed* if $\bot \in A$, $\{s, \neg s\} \subseteq A$ or $s \neq s \in A$ for some term s. Otherwise the branch is *open*. A variable x is *free on a branch* A, if there is a formula $t \in A$ such that $x \in \mathcal{V}t$. Otherwise, x is *fresh* on A.

A *step* is an $n+2$-tuple $\langle A, \mathcal{P}, A_1, ..., A_n\rangle$ of sets of formulas, where $\mathcal{P} \subseteq A$. The branch A is the *head* of the step, the set $\mathcal{P}$ contains the *principal formulae* and each A_i is an alternative, which has the *side formulae* as its members. In the definition of a step in [6] the alternatives are the branches $A \cup A_i$. A *rule* is a set of steps and is usually indicated by a schema such as

$$\mathcal{T}_{BQ} \quad \frac{s =_o t}{s\,,\, t \;\mid\; \neg s\,,\, \neg t}$$

This rule describes the set of steps $\langle A, \{s =_o t\}, \{s, t\}, \{\neg s, \neg t\}\rangle$, where $s =_o t \in A$. A step can be *applied to* a branch B if $\mathcal{P} \subseteq B$ and each $A_i \nsubseteq B$, i.e., the principal formulae are on the branch and the rule's application does not leave the branch unchanged. In this case we also say that the step is *enabled*. A branch B *satisfies* a step, if $\mathcal{P} \cup B$ is closed or $A_i \subseteq B$ for some $i \in \{1, ..., n\}$.

The tableau calculus $\mathcal{T}$ (see Figure 2.2) Satallax uses applies on standard branches.

The set of *$\mathcal{T}$-refutable* branches is defined inductively as follow:

- If A is closed, then A is $\mathcal{T}$-refutable.
- If $\langle A, \mathcal{P}, A_1, ..., A_n \rangle$ is a step and $A \cup A_i$ is $\mathcal{T}$-refutable for all $i \in \{1, ...n\}$, then A is $\mathcal{T}$-refutable.

For our purpose we extend this tableau to the one given in Figure 2.3 by adding rules for the remaining logical constants, transitivity and cut. Further we lift the restrictions from $\mathcal{T}_\forall$ and $\mathcal{T}_\varepsilon$. As $\mathcal{T}$ is a subset of $\mathcal{T}^+$ any $\mathcal{T}$-refutable branch is also $\mathcal{T}^+$-refutable. Any $\mathcal{T}^+$-refutable standard branch is $\mathcal{T}$-refutable by completeness of $\mathcal{T}$. This full tableau calculus $\mathcal{T}^+$ will be used in this thesis for examples and in the implementation. However, this does not necessarily fix the content of $\mathcal{T}^+$. While $\mathcal{T}$ is restricted to aid Satallax by reducing the search space it is safe to include further sound rules in $\mathcal{T}^+$ if it appears useful for the implementation. For convenience if we later mention $\exists$ this will include $\neg\forall$. The same holds for $\forall$ and $\neg\exists$.

2.1.2 Satallax

> Satallax progressively generates higher-order formulas and corresponding propositional clauses. These formulas and propositional clauses correspond to a complete tableau calculus for higher-order logic with a choice operator. Satallax uses the SAT solver MiniSat as an engine to test the current set of propositional clauses for unsatisfiability. If the clauses are unsatisfiable, then the original set of higher-order formulas is unsatisfiable. [2]

An abstract description of the search procedure and implementation of Satallax can be found in [10].

As input Satallax is given a TPTP file in THF format [23] with a list of *definitions, axioms* and an optional *conjecture.* If there is no conjecture Satallax assumes $\bot$ as the conjecture.

Let Atom be a countably infinite set of propositional *atoms.* For each atom a, let $\bar{a}$ denote a distinct negated atom. A *literal* is an atom or a negated atom. Let Lit be the set of all literals. Let $\bar{\bar{a}}$ denote a.

The function $\lfloor . \rfloor$ maps formulae to Lit such that $\lfloor \neg s \rfloor = \overline{\lfloor s \rfloor}$, $\lfloor s \rfloor = \lfloor [s] \rfloor$ and if $[s] = [t]$, then s and t are equivalent. That means a literal is a higher-order normal formula that has been abstracted from all logical connectives, where we only consider whether there is a leading negation. A *propositional clause* is a finite set of literals.

$$\mathcal{T}_\bot\ \frac{\bot}{} \qquad \mathcal{T}_\neg\ \frac{s,\neg s}{} \qquad \mathcal{T}_{\neq}\ \frac{s \neq_\iota s}{} \qquad \mathcal{T}_\rightarrow\ \frac{s \rightarrow t}{\neg s \mid t} \qquad \mathcal{T}_{\neg\rightarrow}\ \frac{\neg(s \rightarrow t)}{s, \neg t}$$

$$\mathcal{T}_\forall\ \frac{\forall_\sigma s}{[st]}\ t \in \mathcal{U}\sigma \qquad \mathcal{T}_{\neg\forall}\ \frac{\neg\forall_\sigma s}{\neg[sx]}\ x \in \mathcal{V}_\sigma \text{ fresh}$$

$$\mathcal{T}_{MAT}\ \frac{\delta s_1 \dots s_n\,,\ \neg\delta t_1 \dots t_n}{s_1 \neq t_1 \mid \dots \mid s_n \neq t_n}\ n \geq 1 \qquad \mathcal{T}_{DEC}\ \frac{\delta s_1 \dots s_n \neq_\iota \delta t_1 \dots t_n}{s_1 \neq t_1 \mid \dots \mid s_n \neq t_n}\ n \geq 1$$

$$\mathcal{T}_{CON}\ \frac{s =_\iota t\,,\ u \neq_\iota v}{s \neq u\,, t \neq u \mid s \neq v\,, t \neq v}$$

$$\mathcal{T}_{BE}\ \frac{s \neq_o t}{s\,, \neg t \ \mid\ \neg s\,, t} \qquad \mathcal{T}_{BQ}\ \frac{s =_o t}{s\,, t \ \mid\ \neg s\,, \neg t}$$

$$\mathcal{T}_{FE}\ \frac{s \neq_{\sigma\tau} t}{\neg[\forall x.sx = tx]}\ x \notin \mathcal{V}_s \cup \mathcal{V}_t \qquad \mathcal{T}_{FQ}\ \frac{s =_{\sigma\tau} t}{[\forall x.sx = tx]}\ x \notin \mathcal{V}_s \cup \mathcal{V}_t$$

$$\mathcal{T}_\epsilon\ \frac{}{[\forall x.\neg(sx)] \ \mid\ [s(\epsilon s)]}\ \epsilon s \text{ accessible, } x \notin \mathcal{V}_s$$

Figure 2.2: Satallax' Tableau calculus $\mathcal{T}$. δ ranges over variables and choice operators. The restrictions for $\mathcal{T}_\forall$ and $\mathcal{T}_\varepsilon$ are described in [6]

There are two special kinds of clauses: *Assumption* and *rule clauses.* Assumption clauses are unit clauses where the single literal is mapped from an axiom or the negation of a conjecture. Rule clauses are derived from a tableau rule (Figure 2.2) and consequently encode a subset of this rule. In a way we can regard rule clauses as the "conjunctive normal form" of the tableau steps.

For example, a clause could be derived from the rule $\mathcal{T}_\rightarrow$ and encode the steps $\langle A, \{s \rightarrow t\}, \{\neg s\}, \{t\}\rangle$ for some formulae s and t and all branches A with $s \rightarrow t \in A$. These steps describe the fact that if $s \rightarrow t$ holds it implies that either $\neg s$ or t holds:

$$\lfloor s \rightarrow t \rfloor \rightarrow \overline{\lfloor s \rfloor} \vee \lfloor t \rfloor \qquad \equiv \qquad \overline{\lfloor s \rightarrow t \rfloor} \vee \overline{\lfloor s \rfloor} \vee \lfloor t \rfloor$$

The resulting rule clause is written as $\overline{\lfloor s \rightarrow t \rfloor} \sqcup \overline{\lfloor s \rfloor} \sqcup \lfloor t \rfloor$.

$$\mathcal{T}_{\bot}\ \frac{\bot}{} \qquad \mathcal{T}_{\neg}\ \frac{s,\ \neg s}{} \qquad \mathcal{T}_{\neq}\ \frac{s \neq_{\iota} s}{} \qquad \mathcal{T}_{\neg\neg}\ \frac{\neg\neg s}{s} \qquad \mathcal{T}_{\rightarrow}\ \frac{s \rightarrow t}{\neg s \mid t}$$

$$\mathcal{T}_{\neg\rightarrow}\ \frac{\neg(s \rightarrow t)}{s, \neg t} \qquad \mathcal{T}_{\vee}\ \frac{s \vee t}{s \mid t} \qquad \mathcal{T}_{\neg\vee}\ \frac{\neg(s \vee t)}{\neg s, \neg t} \qquad \mathcal{T}_{\wedge}\ \frac{s \wedge t}{s, t} \qquad \mathcal{T}_{\neg\wedge}\ \frac{\neg(s \wedge t)}{\neg s \mid \neg t}$$

$$\mathcal{T}_{\forall}\ \frac{\forall_{\sigma} s}{[st]} \qquad \mathcal{T}_{\neg\forall}\ \frac{\neg\forall_{\sigma} s}{\neg[sx]}\ x \in \mathcal{V}_{\sigma}\ \text{fresh} \qquad \mathcal{T}_{\exists}\ \frac{\exists_{\sigma} s}{[sx]}\ x \in \mathcal{V}_{\sigma}\ \text{fresh} \qquad \mathcal{T}_{\neg\exists}\ \frac{\neg\exists_{\sigma} s}{\neg[st]}$$

$$\mathcal{T}_{MAT}\ \frac{\delta s_1 \dots s_n\, ,\ \neg\delta t_1 \dots t_n}{s_1 \neq t_1 \mid \dots \mid s_n \neq t_n}\ n \geq 1 \qquad \mathcal{T}_{DEC}\ \frac{\delta s_1 \dots s_n \neq_{\iota} \delta t_1 \dots t_n}{s_1 \neq t_1 \mid \dots \mid s_n \neq t_n}\ n \geq 1$$

$$\mathcal{T}_{CON}\ \frac{s =_{\iota} t\, ,\ u \neq_{\iota} v}{s \neq u\, ,\ t \neq u \mid s \neq v\, ,\ t \neq v} \qquad \mathcal{T}_{Trans.}\ \frac{s =_{\iota} t\, ,\ t =_{\iota} u}{s = u}$$

$$\mathcal{T}_{BE}\ \frac{s \neq_{o} t}{s\, ,\ \neg t \ \mid\ \neg s\, ,\ t} \qquad \mathcal{T}_{BQ}\ \frac{s =_{o} t}{s\, ,\ t \ \mid\ \neg s\, ,\ \neg t}$$

$$\mathcal{T}_{\leftrightarrow}\ \frac{s \leftrightarrow t}{s\, ,\ t \ \mid\ \neg s\, ,\ \neg t} \qquad \mathcal{T}_{\neg\leftrightarrow}\ \frac{\neg(s \leftrightarrow t)}{s\, ,\ \neg t \ \mid\ \neg s\, ,\ t}$$

$$\mathcal{T}_{FE}\ \frac{s \neq_{\sigma\tau} t}{\neg[\forall x. sx = tx]}\ x \notin \mathcal{V}_s \cup \mathcal{V}_t \qquad \mathcal{T}_{FQ}\ \frac{s =_{\sigma\tau} t}{[\forall x. sx = tx]}\ x \notin \mathcal{V}_s \cup \mathcal{V}_t$$

$$\mathcal{T}_{\epsilon}\ \frac{}{[\forall x. \neg(sx)] \ \mid\ [s(\epsilon s)]} \qquad \mathcal{T}_{Cut}\ \frac{}{s \mid \neg s}$$

Figure 2.3: Full Tableau calculus $\mathcal{T}^{+}$

From some rules we derive more than one clause and only the resulting clauses together encode the steps. For example, if we want to encode the steps $\langle A, \{s =_o t\}, \{s, t\}, \{\neg s, \neg t\}\rangle \subseteq \mathcal{T}_{BQ}$ as propositional clauses, it will result in the clauses $\overline{\lfloor s = t\rfloor} \sqcup \lfloor s\rfloor \sqcup \overline{\lfloor t\rfloor}$ and $\overline{\lfloor s = t\rfloor} \sqcup \overline{\lfloor s\rfloor} \sqcup \lfloor t\rfloor$. Again if $s =_o t$ is true then it implies that either both s and t are true or both are false:

$$\begin{aligned}
& \lfloor s = t\rfloor \rightarrow (\lfloor s\rfloor \wedge \lfloor t\rfloor) \vee (\overline{\lfloor s\rfloor} \wedge \overline{\lfloor t\rfloor}) \\
\equiv\ & \overline{\lfloor s = t\rfloor} \vee (\lfloor s\rfloor \wedge \lfloor t\rfloor) \vee (\overline{\lfloor s\rfloor} \wedge \overline{\lfloor t\rfloor}) \\
\equiv\ & (\overline{\lfloor s = t\rfloor} \vee \lfloor s\rfloor \vee \overline{\lfloor s\rfloor}) \wedge (\overline{\lfloor s = t\rfloor} \vee \lfloor s\rfloor \vee \overline{\lfloor t\rfloor}) \wedge \\
& (\overline{\lfloor s = t\rfloor} \vee \lfloor t\rfloor \vee \overline{\lfloor s\rfloor}) \wedge (\overline{\lfloor s = t\rfloor} \vee \lfloor t\rfloor \vee \overline{\lfloor t\rfloor}) \\
\equiv\ & (\overline{\lfloor s = t\rfloor} \vee \lfloor s\rfloor \vee \overline{\lfloor t\rfloor}) \wedge (\overline{\lfloor s = t\rfloor} \vee \lfloor t\rfloor \vee \overline{\lfloor s\rfloor})
\end{aligned}$$

The $\neg\forall$-rule requires some additional consideration. Let us assume there are the steps $\langle A, \{\neg\forall_\sigma s\}, \{\neg[sx]\}\rangle \subseteq \mathcal{T}_{\neg\forall}$, where besides $\neg\forall_\sigma s \in A$, x has to be fresh in A. This leads us to the clause $\lfloor\forall_\sigma s\rfloor \sqcup \overline{\lfloor sx\rfloor}$ and we say step and clause respectively *select* x as the *witness*. To encode the property of the variable's freshness in the clause the restriction is strengthened. The witness x is globally fresh, i.e., x is not free on any formula that exists at that point in the search.

A *propositional assignment* Φ is a mapping from literals to $\{0, 1\}$, where $\Phi(\bar{l}) = 1 - \Phi(l)$ for every literal l. A propositional clause $\mathcal{C}$ is *satisfied* if there is an assignment Φ such that $\Phi(l) = 1$ for some $l \in \mathcal{C}$. A set of clauses $\mathfrak{C}$ is *propositionally satisfiable* if there is a propositional assignment Φ that satisfies every clause $\mathcal{C} \in \mathfrak{C}$. Otherwise, $\mathfrak{C}$ is *propositionally unsatisfiable.*

2.1.3 SAT Solvers

Whether a set of clauses $\mathfrak{C}$ is propositionally satisfiable or unsatisfiable is a SAT problem. Satallax incrementally calls MiniSat [14] to check $\mathfrak{C}$ for satisfiability. If the set is in the end propositionally unsatisfiable, we will call the utility PicoMus from the SAT solver PicoSat [9] to retrieve the *minimal unsatisfiable core* [5], i.e., a minimal subset of $\mathfrak{C}$ that is still propositionally unsatisfiable.

MiniSat is based on the DPLL algorithm [12] and applies conflict driven learning [19]. We quote MiniSat's basic procedure in Algorithm 1.

Algorithm 1: DPLL with conflict driven learning [14]

```
loop
    propagate()                         // propagate unit clauses
    if not conflict then
        if all variable assigned then
            return SATISFIABLE
        else
            decide()       // pick a new variable and assign it
    else
        analyze()      // analyze conflict and add a conflict
        clause
        if top-level conflict found then
            return UNSATISFIABLE
        else
            backtrack()                        // undo assignments
                                  // until conflict clause is unit
```

2.1.4 Coq

> Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. [1]

A *proof term* is a term in the Calculus of Inductive Constructions. After we give a claim or theorem as a *goal*, Coq starts an interactive environment in which we can use *tactics* to construct a proof for the goal. Applying a tactic can have several consequences:

- It can introduce an *hypothesis* into the *local context*. *Axioms* and *definitions* given before the environment was opened are present in the *global context*.
- New *subgoals* can replace the previous goal. Tactics use backward reasoning, i.e., by proving the new subgoals we can deduce our old subgoal.
- The current subgoal can be *closed*. If all subgoals are closed the proof is complete.

For more detailed information a reference manual [7] and literature [8] on Coq is available.

2.2 Conjecture

The goal of this thesis is to verify the results of Satallax by giving a tableau refutation as a Coq proof script. If Satallax succeeds, we already know that a tableau refutation exists. Hence, to get a refutation we could build an automated higher-order theorem prover that actually searches for the refutation on its own, while simultaneously producing a proof script. However, this would not take advantage of the work Satallax has done. The final state of Satallax will already contain useful information like the set of clauses MiniSat identifies as propositionally unsatisfiable, the universes of instantiation terms and the existential witnesses. We want to make use of this when we construct the refutation.

Therefore we consider that MiniSat actually proves the existence of a refutation, when it confirms the propositional unsatisfiability of the final state of Satallax. If the initial branch A is standard we can show with Proposition 1 in [10] that A is unsatisfiable and thus A is refutable in $\mathcal{T}$ by the completeness of $\mathcal{T}$ (Figure 2.2). Hence we could say that MiniSat indirectly refutes the branch.

Of course, MiniSat does not know $\mathcal{T}$. However, the literals and clauses it has received from Satallax encode higher-order formulae and tableau refutation steps from $\mathcal{T}$. As the set of clauses is finite, MiniSat has only a finite subset of $\mathcal{T}$ available for its refutation, but is still able to show its existence.

Following these thoughts we formulate our conjecture: If Satallax reaches from an initial branch A a *propositionally unsatisfiable* state, we can refute A in the finite subset of $\mathcal{T}$ Satallax has used to create the clause set $\mathfrak{C}$.

Furthermore as the subset is finite, a search for a refutation in this set is guaranteed to terminate.

2.3 Examples

We consider some simple examples to show the effect of our conjecture on a search for a refutation.

2.3.1 The Necessity of Cut

Let us consider the initial branch with the assumptions $\delta s \vee \delta t$, $\neg\delta u \vee \neg\delta t$, $s = t$ and $t = u$, where s, t and u are of type ι and δ is of type ιo.

Using $\mathcal{T}_\vee$ we get the clauses $\overline{\lfloor \delta s \vee \delta t \rfloor} \sqcup \lfloor \delta s \rfloor \sqcup \lfloor \delta t \rfloor$ and $\overline{\lfloor \neg\delta u \vee \neg\delta t \rfloor} \sqcup \overline{\lfloor \delta u \rfloor} \sqcup \overline{\lfloor \delta t \rfloor}$.

Using $\mathcal{T}_{MAT}$ we get the clauses $\overline{\lfloor \delta s \rfloor} \sqcup \lfloor \delta t \rfloor \sqcup \overline{\lfloor s = t \rfloor}$, $\overline{\lfloor \delta t \rfloor} \sqcup \lfloor \delta u \rfloor \sqcup \overline{\lfloor t = u \rfloor}$ and $\overline{\lfloor \delta s \rfloor} \sqcup \lfloor \delta u \rfloor \sqcup \overline{\lfloor s = u \rfloor}$.

Using $\mathcal{T}_{CON}$ we get the clause $\overline{\lfloor s = t \rfloor} \sqcup \lfloor s = u \rfloor \sqcup \overline{\lfloor s = s \rfloor} \sqcup \overline{\lfloor t = u \rfloor}$ among three others.

MiniSat now tells us that the set of clauses is unsatisfiable and we see that we can build a refutation using these steps (see Figure 2.4.a).

But even without the confrontation clauses and the third mating clause the set is still unsatisfiable. Following our conjecture, we should be able to write down a refutation (see Figure 2.4.b).

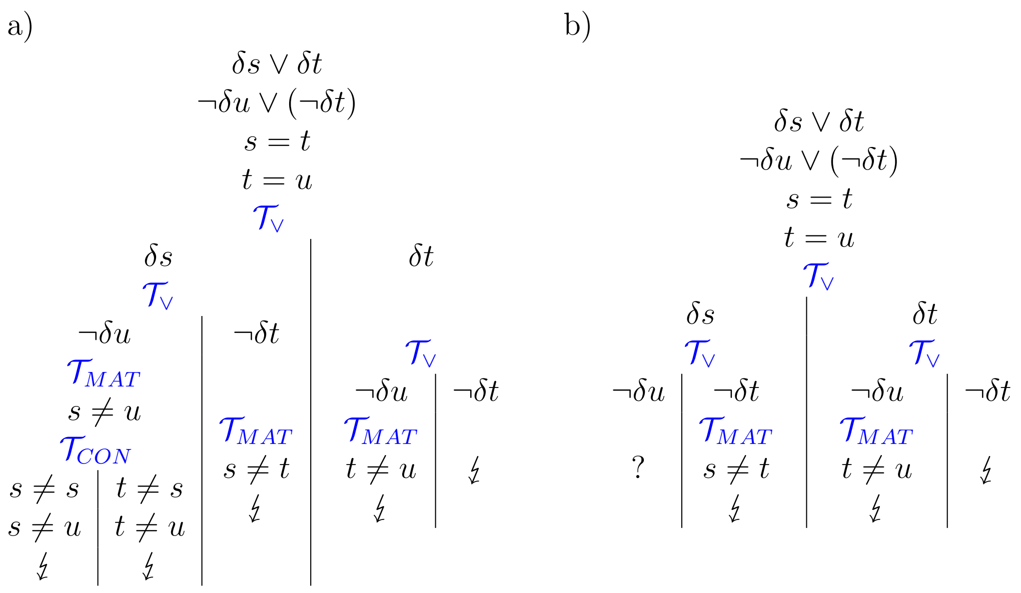

Figure 2.4

For the branch $B_? = \{\delta s \vee \delta t,\ \neg\delta u \vee \neg\delta t,\ s = t,\ t = u,\ \delta s,\ \neg\delta u\}$ none of the encoded steps can be applied anymore, but the branch is still open.

To complete the refutation, one solution is to weaken the condition of our assumption:

We will allow analytic cut to be applied on known terms , e.g., in this case δt (see Figure 2.5.a). We can even reduce the size of the refutation overall by applying the cut early (see Figure 2.5.b).

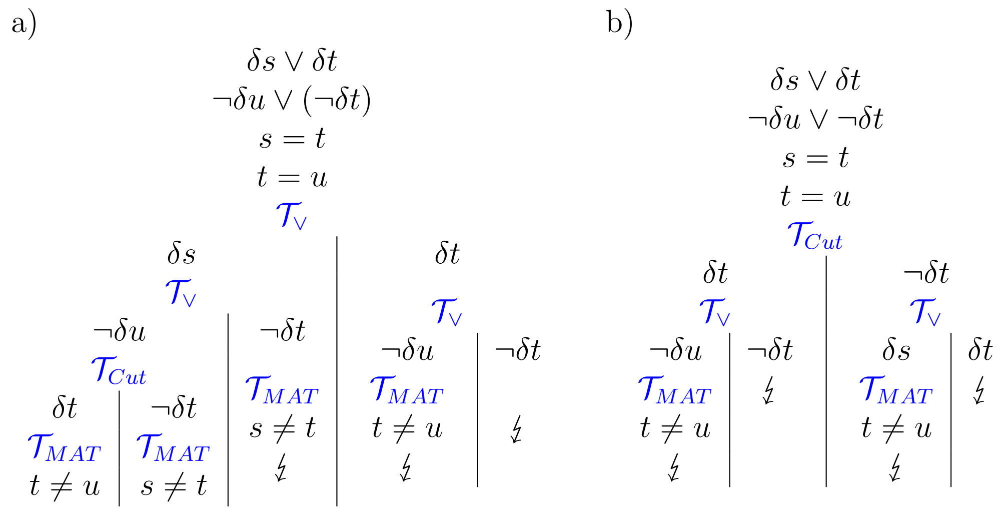

Figure 2.5

2.3.2 The Issue with Freshness

Let us now consider the initial branch with the assumptions $\forall xy.\neg r\ x\ y$ and $(\forall x.r\ x\ x) \vee \exists x.r\ x\ x$, where x and y are of type ι and r is of type $\iota\iota o$.

Using $\mathcal{T}_\vee$ we get the clause $\overline{\lfloor(\forall x.r\ x\ x) \vee \exists x.r\ x\ x\rfloor} \sqcup \lfloor\forall x.r\ x\ x\rfloor \sqcup \lfloor\exists x.r\ x\ x\rfloor$.

Using $\mathcal{T}_\exists$ we get the clause $\overline{\lfloor\exists x.r\ x\ x\rfloor} \sqcup \lfloor r\ a\ a\rfloor$, where a is selected as the witness of type ι.

Using $\mathcal{T}_\forall$ we get the clauses $\overline{\lfloor\forall x.r\ x\ x\rfloor} \sqcup \lfloor r\ a\ a\rfloor$, $\overline{\lfloor\forall xy.\neg r\ x\ y\rfloor} \sqcup \lfloor\forall y.\neg r\ a\ y\rfloor$ and $\overline{\lfloor\forall y.\neg r\ a\ y\rfloor} \sqcup \overline{\lfloor r\ a\ a\rfloor}$.

The resulting set is propositionally unsatisfiable. We can again refute A using these steps (see Figure 2.6.a) However, if we try to apply the steps in a different order, we can get stuck (see Figure 2.6.b).

As a is not fresh on A the restriction of $\mathcal{T}_\exists$ prevents us from applying the step encoded by $\overline{\lfloor\exists x.r\ x\ x\rfloor} \sqcup \lfloor r\ a\ a\rfloor$ to get $r\ a\ a$. However, if we try to solve this by restricting instantiation on variables which are not fresh, we get stuck again (see Figure 2.7.a).

While building the refutation we do not know in general how the refutation of a given branch will be finished. Therefore we risk getting stuck in one of these cases either way and then would have to start backtracking. Again this can be resolved by allowing cut, this time on $\exists x.r\ x\ x$ (see Figure 2.7.b).

On one side we can now introduce a applying the $\exists$-step. On the other side we now have $\neg\exists x.r\ x\ x$ on the branch. As $\exists x.r\ x\ x$ is the only $\exists$-term with a as its witness the branch will be closed before we ever have to consider this $\exists$-step. After finishing the refutation, we can then check whether the $\neg\exists x.r\ x\ x$ was necessary and remove the cut if it was unnecessary.

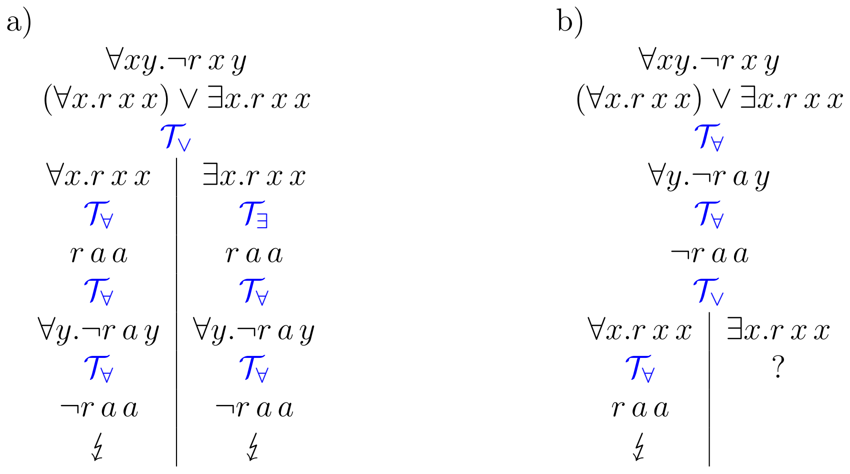

Figure 2.6

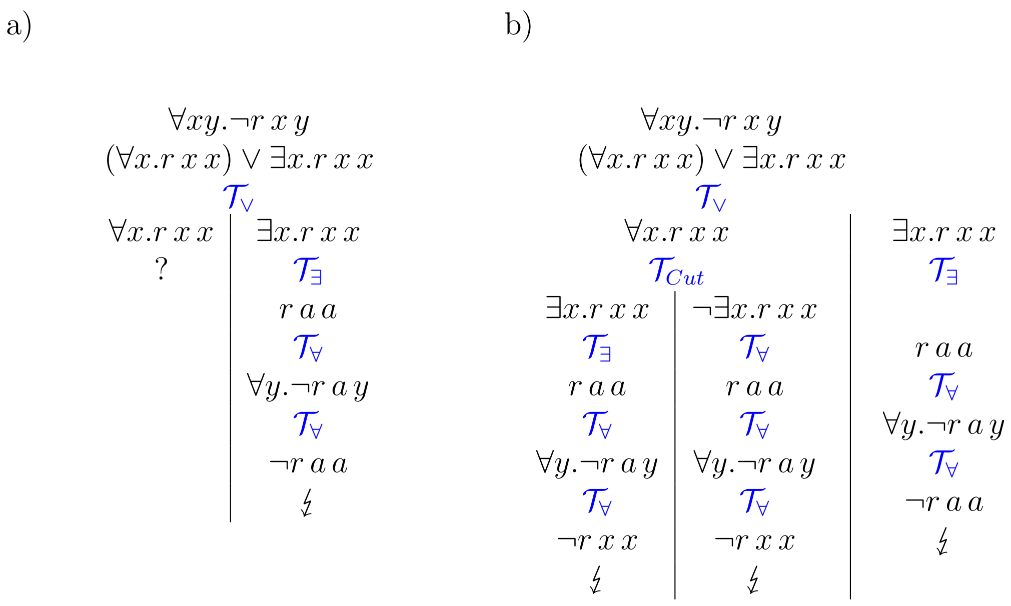

Figure 2.7

3 Theory

3.1 Refutability in F

When Satallax successfully finishes its search, it has found an unsatisfiable set of clauses $\mathfrak{C}$. We define F as the set of all normal formulae present as literals in $\mathfrak{C}$, i.e., $F := \{s \mid s \text{ normal and } \exists \mathcal{C} \in \mathfrak{C}. \lfloor s \rfloor \in \mathcal{C}\}$. By our conjecture this set of known formulae is sufficient to construct a refutation. In the following chapter we will make this conjecture precise and give a constructive proof for it.

For simplicity we use standard formulae and the restricted tableau calculus $\mathcal{T}$ (Figure 2.2). In the implementation Satallax uses rewriting in a preprocessing phase for standardisation. Furthermore, we assume that for every clause $\mathcal{C} \in \mathfrak{C}$, which partly encodes a step, all clauses encoding that step are in $\mathfrak{C}$. Otherwise we add them.

3.2 Definitions

As a SAT solver handles every formula as a sign free atom, we will consider for every formula its negation, too. To prevent creating infinitely many negations of negations we normalize every formula s using the equivalence $\neg\neg s \equiv s$. This way there will be at most one negation in front of every formula.

Definition 3.2.1 (negation)**.** *Let s be a formula,*
then $\bar{s} = \begin{cases} t, & \textit{if there is a } t \textit{ such that } s = \neg t \\ \neg s, & \textit{otherwise} \end{cases}$
A finite set of formulae F is closed under negation, if for every $s \in F, \bar{s} \in F$.

To show that a set of clauses is propositionally unsatisfiable we need to show that every propositional assignment Φ does not satisfy all clauses. In our set of higher-order formulae we will simulate Φ as a full expansion of the initial assumptions, i.e., we add as many formulae from our known set F to the branch as possible without closing it.

This way we will have a branch that contains every formula or its negation. For such a full expansion B there is an assignment $\Phi(\lfloor s \rfloor) := \begin{cases} 1 & \text{if } s \in B \\ 0 & \text{otherwise} \end{cases}$.

Definition 3.2.2 (full expansion)**.** *Let A be an open branch and F a finite set of normal formulae, where $A \subseteq F$. We call B a full expansion if $A \subseteq B \subseteq F$, B is open and $\forall s \in F,\ s \in B$ or $B \cup \{s\}$ is closed.*

Making use of the finiteness of F we restrict our tableau calculus $\mathcal{T}$, i.e., we only consider steps in $\mathcal{T}$ where all formulae are in the negative closure of F. This way the restricted calculus $\tilde{\mathcal{T}}$ will be finite as well. We will further assume that we know for every type σ the universe of instantiations $\mathcal{U}\sigma$. Knowing $\mathcal{U}$ makes testing whether a certain formula is the result of a $\forall$-instantiation decidable. This information can be extracted from Satallax. As discussed in the previous chapter we also require analytic cuts and thus add for every formula and its negation in F the corresponding cut step.

Definition 3.2.3 (tableau calculus $\tilde{\mathcal{T}}$)**.** *Let F be a finite set of normal formulae
and $\mathcal{U}$ a finite set of terms .*

$$\begin{array}{rcl} \tilde{\mathcal{T}}_F^{\mathcal{U}} & := & \{\langle A, \mathcal{P}, A_1, ..., A_n \rangle \in \mathcal{T} \setminus \mathcal{T}_\forall \mid A \cup A_1 \cup .. \cup A_n \subseteq F\} \\ & \cup & \{\langle A, \{\forall_\sigma s\}, \{[st]\}\rangle \mid A \cup \{[st]\} \subseteq F \textit{ and } t \in \mathcal{U}\} \\ & \cup & \{\langle A, \emptyset, \{s\}, \{\neg s\}\rangle \mid A \cup \{s, \neg s\} \subseteq F\} \end{array}$$

We often write $\tilde{\mathcal{T}}$ when F and $\mathcal{U}$ are clear in context.

We already saw in an example that we have to consider the freshness of existential witnesses when applying an $\exists$-step. Therefore we restrict the use of certain variables. In general a fresh variable can be freely introduced to a branch by an $\neg\forall$-, a $\forall$- or a cut-step. A *blocked* variable however can be only introduced by an $\neg\forall$-step. As Satallax chooses a new globally fresh witness every time it adds an $\neg\forall$-clause to the set, we again extract this information during runtime and represent it as the *selecting function* S, which maps witnesses to their selecting formulae. To be able to decide in a refutation which variables are blocked and in what order they can be introduced, we define a binary relation $<_S$ on the domain of S [21, Def.3.2]. For two variables x and y, $x <_S y$ holds, if x is free in the selecting formula of y, i.e., $\neg\forall t$, where $t := S(y)$. This relation describes the dependencies between blocked variables and, if it is acyclic, will describe their order. From this definition it also follows that $<_S$ is irreflexive as $x <_S x$ does not hold for any variable x.

Definition 3.2.4 (relation $<_S$). *For a function S from variables to terms, $<_S$ is the binary relation on variables in dom S,*
where for every $x, y \in$ dom $S, x <_S y \Leftrightarrow x$ is free in $S(y)$.
The function S^{-x} : dom $S \backslash \{x\} \to$ ran S is defined by $\forall y \in$ dom $S^{-x}, S^{-x}(y) = S(y)$.

Now we can give an abstract description of the result of Satallax. Called on the initial assumptions A, Satallax returns the set of formulae F, the selecting function S and the universe of instantiation terms $\mathcal{U}$. These together are called an abstract refutation of A if they fulfil the following sufficient conditions to prove refutability of A:

- The relation $<_S$ has to be acyclic and every variable in the domain of S has to be fresh in A. This ensures that there is an order in which all witnesses are fresh when instantiated without having cyclical dependencies between them. This is given by the way Satallax chooses globally fresh witnesses and their chronological order is a linearisation of $<_S$.

- Every full expansion B of A either has to be refutable in one step or there is an x in the domain of S such that its selecting formula and the negation of its instantiation are on B. The first part means that if, e.g., for some formulae s and t $s, \neg t$ and $s \to t$ are on B, we can refute B just by applying the tableau rule $\mathcal{T}_{\to}$. The second part describes the case where we would like to close B with $\mathcal{T}_{\neg\forall}$. As $[tx]$ is on the branch, x is not fresh anymore and the condition of $\mathcal{T}_{\neg\forall}$ is violated. However we presume that there is a way to prevent this case.

The second condition is equivalent to the set of clauses $\mathfrak{C}$ being propositionally unsatisfiable, which was verified by MiniSat. As every full expansion B also represents a propositional assignment Φ, there is at least one clause $\mathcal{C} \in \mathfrak{C}$ Φ does not satisfy. Hence B does not satisfy the step encoded by $\mathcal{C}$. As B is a full expansion this step is enabled and its application closes the branch.

For example, let us assume the mating clause $\overline{\lfloor p\ s \rfloor} \sqcup \lfloor p\ t \rfloor \sqcup \overline{\lfloor s = t \rfloor}$ is not satisfied by the full expansion B. Thus $\{p\ s, \neg p\ t, s = t\} \subseteq B$. As the principal formulae $p\ s$ and $\neg p\ t$ are on B, we can apply $\mathcal{T}_{MAT}$ to refute B by showing that $B \cup \{s \neq t\}$ is closed. This is the case because $s = t$ and $s \neq t$ are conflicting.

In a second example let us assume the clause $\lfloor s = t\rfloor \sqcup \lfloor s\rfloor \sqcup \lfloor t\rfloor$ is unsatisfied by B. Thus $\{s \neq t, \neg s, \neg t\} \subseteq B$. The boolean extensionality step $\langle B, B \cup \{s, \neg t\}, B \cup \{\neg s, t\}\rangle$ is encoded by this clause along with $\lfloor s = t\rfloor \sqcup \overline{\lfloor s\rfloor} \sqcup \overline{\lfloor t\rfloor}$. As the principal formula $s \neq t$ is on B, we can apply $\mathcal{T}_{BE}$ to refute B by showing that $B \cup \{s, \neg t\}$ and $B \cup \{\neg s, t\}$ are closed. Again there is a conflict in both cases closing the branch.

Definition 3.2.5 (abstract refutation)**.** *Let A be an open branch,*
F a finite set of normal formulae closed under negation,
$\mathcal{U}$ a finite set of terms and S a function from variables to terms.
Then we call $(F, S, \mathcal{U})$ an abstract refutation of A, if

1. *$<_S$ is acyclic (or a strict partial order).*

2. *For every $x \in dom\ S$, x is not free in A and $\neg\forall t \in F$ and $\neg[tx] \in F$, where $t = S(x)$.*

3. *For every full expansion B, either*
   B is refutable in $\tilde{\mathcal{T}}_F^{\mathcal{U}}$ in one step or
   there is an $x \in dom\ S$ such that $\neg\forall t \in B$ and $[tx] \in B$ where $t = S(x)$.

This also has an important consequence for the implementation. We can often considerably reduce the size of the abstract refutation by taking the minimal unsatisfiable core of the resulting clause set. As this can only make F,S and $\mathcal{U}$ smaller. This has the consequence that the $<_S$ relation can only become less restrictive and thus the first and second condition are fulfilled. The UNSAT core is still propositionally unsatisfiable, which is equivalent to the third condition as shown above.

3.3 Proof

First we will prove our conjecture with a stronger condition and use this result as a lemma for the real theorem. For this lemma we assume that we have an abstract refutation of our initial problem A, but there are no existential witnesses and thus the selecting function S is empty. We give a constructive proof of the existence of a refutation by induction. We apply cut on all formulae in F until every open branch in the refutation is a full expansion of the initial branch A. Then we can close all of them in one step, as the one problematic case of applying $\mathcal{T}_{\neg\forall}$ never occurs.

Lemma 3.3.1. *If $(F, \emptyset, \mathcal{U})$ is an abstract refutation of A, then A is refutable in $\tilde{\mathcal{T}}_F^{\mathcal{U}}$.*

Proof. By induction on the distance of A from a full expansion:
In the base case A is a full expansion. As $(F, \emptyset, \mathcal{U})$ is an abstract refutation of A, by Definition 3.2.5 A is refutable in $\tilde{\mathcal{T}}$ in one step since $dom\ \emptyset$ is empty. Therefore, A is refutable in $\tilde{\mathcal{T}}$.
As the inductive hypothesis we now assume that for every open branch A', where $A \subset A' \subseteq F$ and $(F, \emptyset, \mathcal{U})$ is an abstract refutation of A', A' is refutable in $\tilde{\mathcal{T}}$.
In the induction step A is not a full expansion of itself. Hence there is a $t \in F$ such that $t \notin A$ and $\bar{t} \notin A$.
By use of Cut on t it is enough to show that A, t and $A, \neg t$ are refutable in $\tilde{\mathcal{T}}$.
As every full expansion of A, t or $A, \neg t$ is a full expansion of A as well, $(F, \emptyset, \mathcal{U})$ is an abstract refutation of A, t and $A, \neg t$. Thus by inductive hypothesis, A, t and $A, \neg t$ are refutable in $\tilde{\mathcal{T}}$. □

Now we can prove the real theorem: Given an abstract refutation we can build a refutation in the finite tableau calculus $\tilde{\mathcal{T}}$. We again use a proof by induction, where our base case is the lemma we just proved and in each induction step we reduce the size of S. In each step we introduce one blocked witness x by applying cut on its selecting formula $\neg\forall t$. This creates two new branches: One with $\neg\forall t$ and another with $\forall t$. On the former we apply $\mathcal{T}_{\neg\forall}$. On the latter we can freely use x, because the branch will be already closed if we need x as a witness. Thanks to the strict partial order $<_S$ we can choose this witness as a minimal element in the domain of S and thereby prevent accidentally introducing other witnesses with the cut.

Theorem 3.3.2. *If $(F, S, \mathcal{U})$ is an abstract refutation of A, then A is refutable in $\tilde{\mathcal{T}}_F^{\mathcal{U}}$.*

Proof. By induction on $|dom\ S|$:
In the base case $dom\ S$ is empty. Hence, $(F, \emptyset, \mathcal{U})$ is an abstract refutation of A and by Lemma 3.3.1, A is refutable in $\tilde{\mathcal{T}}$.
As the inductive hypothesis we now assume that for every pair (A', S'), where $A \subseteq A' \subseteq F$, $S' \subset S$ and $(F, S', \mathcal{U})$ is an abstract refutation of A', A' is refutable in $\tilde{\mathcal{T}}$.
In the induction step $dom\ S$ is not empty. Therefore there is an $x \in dom\ S$, which is a $<_S$-minimal element and for convenience we define $t := S(x)$.
By use of Cut on $\forall t$, it is enough to show that $A, \forall t$ and $A, \neg\forall t$ are refutable in $\tilde{\mathcal{T}}$. By Definition 3.2.4 and 3.2.5, x is not free in either A or t.

Thus, by use of $\tilde{\mathcal{T}}_{\neg\forall}$, it is enough to show that $A, \forall t$ and $A, \neg\forall t, \neg[tx]$ are refutable in $\tilde{\mathcal{T}}$.
To apply the inductive hypothesis, $(F, S^{-x}, \mathcal{U})$ can be used as an abstract refutation of $A, \forall t$ and $A, \neg\forall t, \neg[tx]$ because:

1. As $<_S$ is acyclic, $<_{S^{-x}}$ is acyclic as well.

2. For every $y \in \mathit{dom}\ S$ such that $y \neq x$, y is free neither in A by definition 3.2.5.2 nor in t, as $y \not<_S x$. As $\mathcal{V}[s] \subseteq \mathcal{V}s$, y is also not free in $\neg[tx]$. Ergo, for every $y \in \mathit{dom}\ S^{-x}$, y is not free in either $A, \forall t$ or $A, \neg\forall t, \neg[tx]$.

3. As every full expansion B of $A, \forall t$ or $A, \neg\forall t, \neg[tx]$ is a full expansion of A as well, either B is refutable in $\tilde{\mathcal{T}}$ in one step or there is a $y \in \mathit{dom}\ S$ such that $\neg\forall s \in B$ and $[sy] \in B$ where $s = S(y)$. As B is open and either $\neg[tx] \in B$ or $\forall t \in B$, either $\neg\forall t \notin B$ or $[tx] \notin B$. Thus, for every full expansion B, either B is refutable in $\tilde{\mathcal{T}}$ in one step or there is a $y \in \mathit{dom}\ S^{-x}$ such that $\neg\forall s \in B$ and $[sy] \in B$ where $s = S(y)$.

Now, $A, \forall t$ and $A, \neg\forall t, \neg[tx]$ are by inductive hypothesis refutable in $\tilde{\mathcal{T}}$. $\square$

With this we have proven our conjecture that we can find a refutation for a branch only using the formulae and consequently the refutation steps Satallax has used in its search.

From another point of view, we can see Theorem 3.3.2 as a kind of completeness proof for our restricted tableau calculus $\tilde{\mathcal{T}}$. While it is not complete in general, it is guaranteed to refute A.

The proof also describes one way to construct such a refutation. In a first phase we introduce all existential witnesses before reconstructing a resolution refutation like a SAT solver would produce and simultaneously translating it one to one into a tableau refutation.

Unfortunately this way of doing it is not very practical as the consecutive cuts produce an exponential number of branches. Even if we apply closing rules early and subsequently remove redundant branches and thereby may reach reasonable sized refutations, this would still not be an intuitive tableau refutation.

In the beginning cut was not even part of the tableau calculus. We just allowed it as a compromise to complete a refutation in certain special cases, but now it is almost the only rule applied. All other rules are ignored except at the beginning and for the leaves of the refutation. It does not make use of either non-branching rules like $\mathcal{T}_\forall$ and $\mathcal{T}_{\neg\rightarrow}$ or rules producing more than one result at once like $\mathcal{T}_{CON}$ and $\mathcal{T}_{BQ}$ to reduce the number of branches (see Figure 3.1).

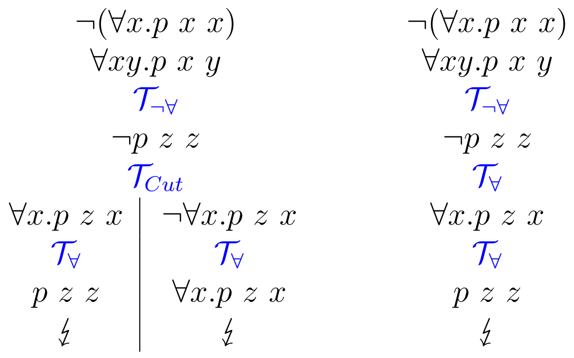

Figure 3.1: Comparison between two refutations. The left is produced following the method proposed by Lemma 3.3.1 and Theorem 3.3.2, while in the refutation on the right the steps are applied in an intuitive order.

Therefore we have chosen a different approach for the actual implementation, which produces more intuitive and often shorter refutations. However, it still follows the initial idea the proof is build on: Applying – for the most part – only the rules Satallax has used to create the propositionally unsatisfiable set of clauses.

4 Implementation

The central idea for the algorithm is that we only need the refutation steps Satallax has used to create the clauses. We refute the initial branch by consecutively applying these steps to the resulting branches until all of them are closed. As the set of clauses is finite and unsatisfiable, any branch, where all steps encoded in the clauses have been applied, will be closed and the algorithm is guaranteed to finish eventually. As seen in the first example this alone can sometimes get stuck when there are still unused steps but none of them can be applied. In such cases we will use the algorithm described by the lemma and theorem in Section 3.3 as a fallback and apply cut to introduce missing formulae to the branch.

This approach allows us to take advantage of the freedom to choose the next step, when searching for a refutation, as long as we are not introducing existential witnesses with rules other than $\mathcal{T}_\exists$. This way we can influence the structure of the resulting refutation.

We give here an informal description of the algorithm:

We are given a set of steps and the branch we want to refute. We first remove all steps that are satisfied by the branch. From the remaining steps we choose one for the current branch. If this step can be applied and respects the restriction on fresh witnesses, we do so and recursively call the algorithm on the new branches with the reduced set of steps. Otherwise we use cut to lift the restriction on a witness or introduce a new formula to the branch and again continue recursively.

This describes the first part of the implementation, where a refutation is extracted from the final state of Satallax. As Satallax standardises logical constants, the starting branch will be different from the initially given higher-order problem and the refutation will have holes where Satallax has applied its normalization during the search (see Section 4.3.3). The following translation will consider the normalizations to fill those holes in the refutation. In the last part the specified output is created. In our case this is a Coq proof script and additionally we then account for certain technical details of Coq.

4.1 Search

The core mechanic of the search is divided into two parts: OR- and AND-search. While the former chooses from all available steps one to be applied next, the latter applies this step and continues the search on all resulting branches. All other functionality is divided between them accordingly.

4.1.1 OR-search

Algorithm 2: or_search (**Branch** B, **List**⟨**Step**⟩ *steps*)

```
Output: Refutation
if $branch\_is\_closed(B)$ then
    return Refutation.$conflict(B)$
else
    $steps \leftarrow remove\_satisfied\_steps(B, steps)$
    $t \leftarrow get\_next\_step(B, steps)$
    return $and\_search(B, t, steps)$
```

In the beginning OR-search is called on the branch containing only the initial assumptions and a list of steps derived from the clauses found by Satallax.

The first thing OR-search does is checking whether the given branch is closed. For this, the old branch and the newly added formulae are given separately and then merged one by one. Should one of them be $\perp$, a negation of a trivial equation, e.g., $s \neq s$ for some term s, or having its negation already on the branch, a corresponding leaf of the refutation is returned.

If this is not the case, the next step will be to remove satisfied steps from the list. As a reminder a step will be satisfied, if its application leaves at least one branch unchanged or if the negation of one of its principal formulae is on the branch, which means that the branch will be closed anyway when this step becomes applicable. As this is equivalent with the propositional assignment of the branch satisfying the clause encoding the step, we say it is satisfied.

Following on this, a step is chosen from the remaining list. As described in the proof, the only condition for choosing this clause is that we are not introducing any existential witnesses which might need to be fresh later in the refutation. We prevent this by marking variables as blocked if there is an $\exists$-step on the list selecting it as a witness. A blocked variable can only be introduced by its selecting step. On the other hand this means that if there is no such step or it has been removed, the variable can be freely used. Note, that choosing the steps in the order Satallax produced the corresponding clauses fulfils this condition.

In the end AND-search is called with the updated branch, step list and chosen step.

4.1.2 AND-search

Algorithm 3: and_search(**Branch** B, **Step** t, **List**⟨**Step**⟩ *steps*)

```
Output: Refutation
if $can\_apply(B, t)$ then
    $subbranches \leftarrow apply\_step(B, t)$
    $sub\_refutations \leftarrow []$
    for $subbranch$ in $subbranches$ do
        $sub\_refutations \leftarrow or\_search(subbranch, steps) ::$
        $sub\_refutations$
    return Refutation.$make(t.get\_rule(), sub\_refutations)$
else
    $c \leftarrow get\_missing\_principal(B, t)$
    $left \leftarrow or\_search(B \cup \{c\}, steps)$
    $right \leftarrow or\_search(B \cup \{\overline{c}\}, steps)$
    return Refutation.$make($Cut$, [left, right])$
```

The behaviour of AND-search depends on whether the given step can be applied to the branch. As it is assumed that the freshness of witnesses was preserved, the only thing to check is whether the step's principal formulae are on the branch.

If this is the case, we apply the step and recursively call OR-search on every new branch. We then combine the resulting refutations into one refutation node corresponding to the applied step.

In the negative case, there is at least one principal formula which is not on the branch. On this formula we apply cut. This gives us two new branches: One of them has the missing principal formula and if it was the only one missing, we can apply the step should it be chosen again. As the other branch contains the negation of the step's principal formulae, it is now satisfied and will be removed. After this we continue the same way as in the first case by refuting both branches and creating a new refutation node.

4.2 Refinements

With these two functions calling each other recursively we have a simple working core for our search. In the following we will introduce refinements to increase speed and reduce the size of the result.

4.2.1 Preprocess

Before we start the search, there are several things we can prepare to improve performance.

UNSAT-core

As mentioned in a remark at the end of 3.2 the algorithm will still work, if we replace the set of clauses from the final state by a minimal UNSAT core. For an efficient generation of the core, we use PicoMus [9], because MiniSat does not have this functionality. In our tests at least 50% of the clauses are removed, but usually far more and in some cases more than 99%.

Turn Clauses into Steps

Algorithm 4: class **Step**

```
Rule rule
List<int> principal_formulae
List<List<int>> alternatives
List<string> witnesses
Step make(List<int> clause, List<string> global_witnesses)
bool satisfied(Branch B)
```

To avoid redundant computations during the search we first turn the propositional clauses into a more general data structure: Steps. They are not the same as tableau steps, but sets of tableau steps. While the principal formulae and side formulae are fixed, the actual branch will be taken from the context of the search. This means that in the AND-search branch B and step t will together define a single tableau step. Additionally, the step structure contains precomputed information about existential witnesses appearing free in the step. As Satallax produces for some rule more than one clause, e.g., two for $\mathcal{T}_{\neg\rightarrow}$ and four for $\mathcal{T}_{CON}$, changing to steps reverts these clauses back into one step and we only need one of those clauses to reconstruct the step.

Adding Symmetric Steps

After we have removed a lot of clauses by computing a minimal core, this preprocessing step appears counter-intuitive at first. With an exponential worst case complexity depending on the number of steps one might think, that the search is faster with as few steps as possible. However, in some test cases the refutation becomes smaller when we put certain steps back that have been removed. Although the algorithm is still able to find a refutation, with fewer steps to choose from it has a higher chance to need a cut in order to enable a rule. This sometimes leads to more subtrees and in general a larger refutation. In these special problems the solution was to add for every mating step the symmetric version, i.e., we switch the signs of the principal formulae. We give an example in Section 5.2. This can be done with the confrontation rule as well. However, these new steps do not increase the worst case runtime, because one will be satisfied if the other is enabled. Thus at most one of them can be applied in a given branch.

Sorting

The last preprocessing step is to sort the list of steps. We order the steps following three criteria:

At the beginning of the list the $\exists$-steps are chronologically ordered, which is the same order they have been in before. This will be required by the procedure choosing the steps.

The remaining steps are in an ascending order by the number of subgoals their application creates. As we need a refutation for every subgoal, it is usually better to first apply rules with fewer subgoals. For example, if we have a step with two and one with three alternatives and need to apply both, in the worst case we get six branches, but in one case we have applied three steps (see Figure 4.1.a)) and in the other four (see Figure 4.1.b)). Of course, if two branches were closed after applying the three-alternative step, we would only need two steps (see Figure 4.1.c)). However as we cannot know this without trying both ways, we choose the first as the worst case yields a smaller refutation.

The last criterion is the number of the side formulae occurrences on other steps. Steps with the same number of alternatives are sorted by this criterion in descending order. This way steps with a larger impact on all other steps, either satisfying or enabling them, come first in the list.

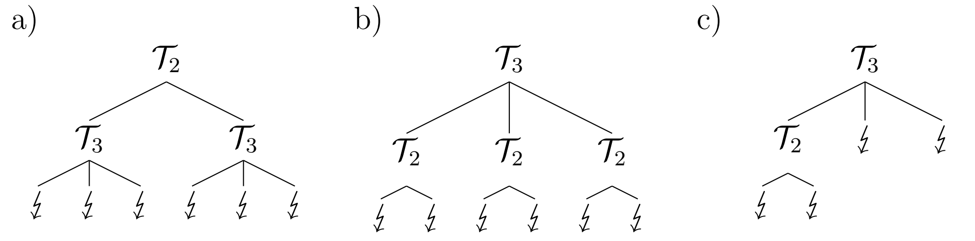

Figure 4.1: Three possible refutations with two steps: $\mathcal{T}_2$ with two alternatives and $\mathcal{T}_3$ with three alternatives.

4.2.2 Heuristic

We already described which conditions the procedure that chooses the next step in OR-search must fulfil. However these restrictions are rather weak and leave us with a lot of choices. Many algorithms use heuristics for such decisions. Actually the main part of our heuristic is already describe in Section 4.2.1.

When we call the procedure that chooses the next step, we assume that the given list contains no steps which are satisfied by the current branch and that all $\exists$-steps are at the head of the list. We then iterate over the list until we find a suitable step, i.e., the step is enabled and does not contain a blocked variable. At the beginning the set of blocked variables is empty. For every unsuitable $\exists$-step we add its witness to the set. This way we make sure that the witnesses of $\exists$-steps that could be applied later stay fresh. If we arrive at the end of the list without finding a suitable step, the first step in the list will be returned. While there are still $\exists$-steps this results in a forced introduction of the witness via cut.

In this approach the position in the list decides which step among the suitable steps is chosen. As the search does not change the order of the list the initial sorting determines the chosen step. We also tried in the implementation to apply dynamic heuristics which consider the current branch. However, while they required a significant amount of computation, their effect on the result was negligible. The most reasonable explanation for this is that the steps left in the unsatisfiable core at the beginning are approximately equally important for the refutation. Also, there are often not more than a few steps suitable at a time. Thus the order in which they are applied is unimportant.

4.2.3 Tracking Dependencies

As we cannot know in advance how the branches of a certain rule application will be refuted, it is very likely that we apply unnecessary steps during the search. However, afterwards we can at least check whether a step was necessary. If one subrefutation does not use the side formulae added in that step, we can remove the step and instead use this independent refutation to refute the branch. If we do this during the search each time OR-search returns a refutation, we will even save work in case the step had more open branches.

The implementation handles this by adding to a refutation its dependency. For a leaf of the refutation those are the literals that directly cause the branch to be closed. For a node the dependency is the union of the step's principal formulae and the dependencies of its subrefutations, where the corresponding side formulae have been removed. This way we get for a node the subset of the current branch that has indirectly caused it to be closed.

In a propositional context we can see dependencies as learnt clauses [19], where we are backtracking tableau steps instead of backtracking propagation. Following this comparison suggests to implement learning as well, which we leave for future work.

4.2.4 Semantic Branching

In the example in Section 2.3.1 we not only show that cut is in special cases necessary, but can also reduce the size of the result. Although our implementation still tries to avoid cut when possible, as it might make things worse, semantic branching [11] gives us a tool to use cut to enhance a regular step.

For example, assume we have an implication $s \to t$ on the branch. Instead of applying $\mathcal{T}_{\to}$ we first use cut on s and then apply $\mathcal{T}_{\to}$ on the first subgoal. This results in three branches (see Figure 4.2 case 4). The first is closed as it contains both s and $\neg s$. The third branch with $\neg s$ already satisfies the $\to$-step. The second branch now contains t and s instead of just t. This means should later in the refutation a step have $\neg s$ as a side formula this branch will be closed. Otherwise we would in the worst case have to repeat the whole refutation of our third branch. The same argument works analogously for a cut on t.

By using dependency tracking in Section 4.2.3 we can even try both semantic and the syntactic rule at the same time while only adding constant overhead:

In our example we first refute the branch with t and s added. If s is in the dependency of this refutation, we use semantic branching with cut on s. Otherwise, we refute the branch with $\neg s$ and $\neg t$ added. If $\neg t$ is in the dependency, we use cut on t. If not, then we use just syntactic branching. Of course, there are more cases which are considered in Algorithm 5 and Figure 4.2.

In our tableau calculus semantic branching can be used in a similar way with the confrontation rule (Figure 4.3), mating rule and decomposition rule (Figure 4.4), too.

Algorithm 5: semantic branching for implication

```
Input: Term s, Term t and Branch B, where $s \to t \in B$
Output: Refutation

if $s \in B$ or $\neg t \in B$ then
    Case 1: syntactic branching (Figure 4.2)
else
    $(\mathcal{R}_1, D_1) \leftarrow or\_search(B \cup \{s, t\})$
    if $s \in D_1$ then
        $(\mathcal{R}_2, D_2) \leftarrow or\_search(B \cup \{\neg s\})$
        if $t \notin D_1$ and $\neg s \in D_2$ then
            Case 2: only cut on $s$ (Figure 4.2)
        else if $t \in D_1$ and $\neg s \in D_2$ then
            $(\mathcal{R}_2, D_2) \leftarrow or\_search(B \cup \{\neg s\})$
            Case 4: semantic branching with cut on $s$ (Figure 4.2)
        else                                                  /* $\neg s \notin D_2$ */
            Case 7: refutation $\mathcal{R}_2$ is independent
    else if $s \notin D_1$ and $t \in D_1$ then
        $(\mathcal{R}_2, D_2) \leftarrow or\_search(B \cup \{\neg t, \neg s\})$
        if $\neg t \notin D_2$ and $\neg s \in D_2$ then
            Case 1: syntactic branching (Figure 4.2)
        else if $\neg t \in D_2$ and $\neg s \notin D_2$ then
            Case 3: only cut on $t$ (Figure 4.2)
        else if $\neg t \in D_2$ and $\neg s \in D_2$ then
            Case 5: semantic branching with cut on $t$ (Figure 4.2)
        else                                  /* $\neg t \notin D_2$ and $\neg s \notin D_2$ */
            Case 7: refutation $\mathcal{R}_2$ is independent
    else                                          /* $s \notin D_1$ and $t \notin D_1$ */
        Case 6: refutation $\mathcal{R}_1$ is independent
```

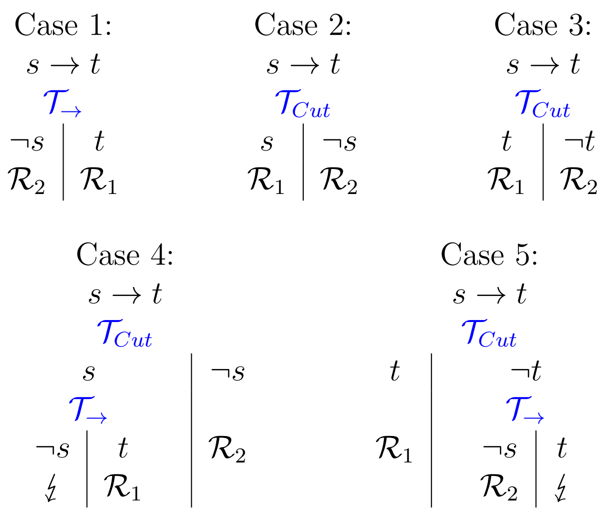

Figure 4.2: Case 1: Syntactic branching; Case 2: Cut on s; Case 3: Cut on t; Semantic branching with... (Case 4: ...cut on s; Case 5: ...cut on t)

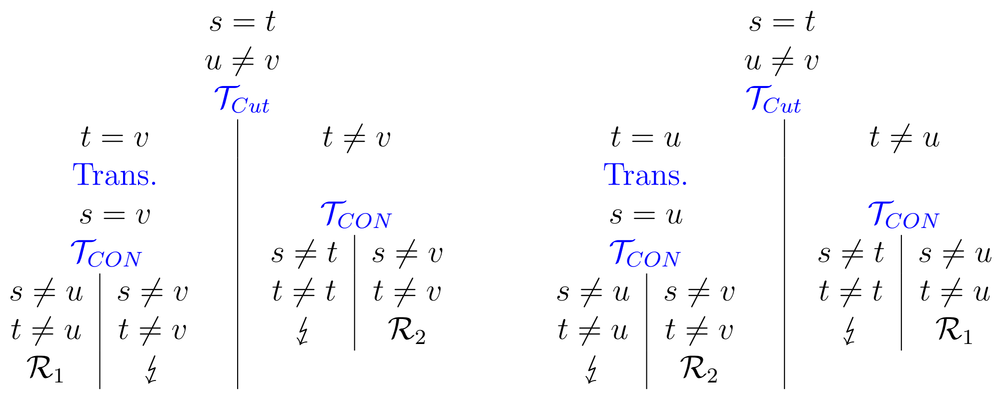

Figure 4.3: Semantic branching for confrontation. Although strictly they are not steps in $\tilde{\mathcal{T}}$ we use transitivity of equality and an additional confrontation step. Transitivity could be replaced by another confrontation step at a later point in the refutation, but it would creates more branches and could be required more often than an immediate application of transitivity.

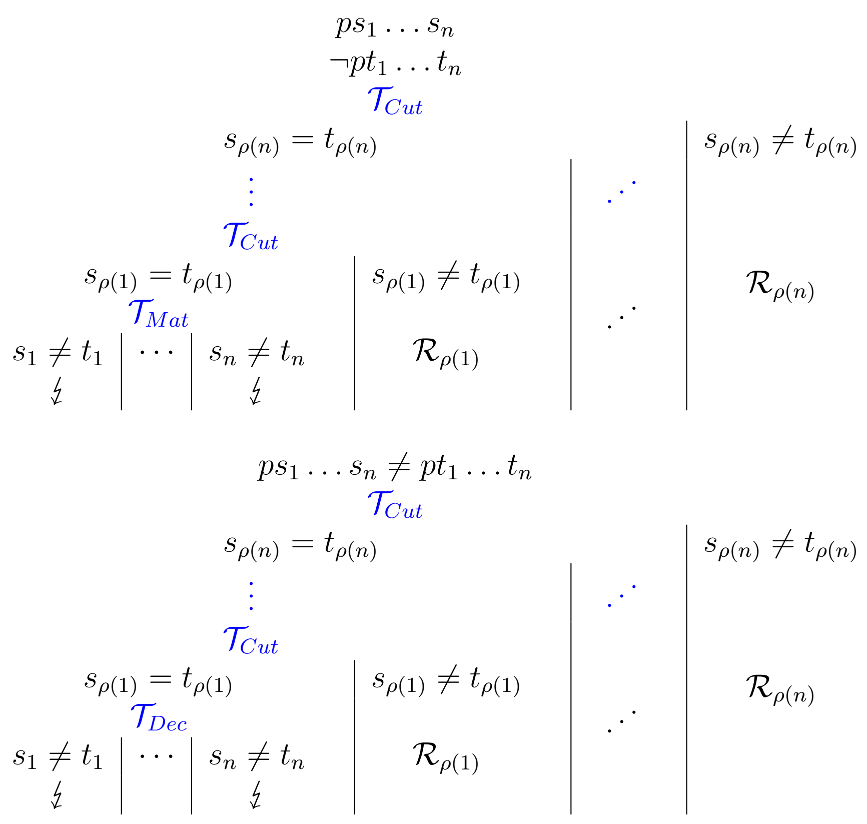

Figure 4.4: Semantic branching for mating and decomposition. ρ is a permutation of $\{1..n\}$ depending on the resulting conditions of the subrefutations. The final step can be shortened by removing cuts for unused dependencies.

4.3 Translation

After we have found a refutation in the first part of our implementation the next step is to apply a translation, which considers the standardisation and normalization Satallax uses. The goal of this is to turn the refutation we found into one that matches the initial problem and explicitly states the used rewrite steps.

4.3.1 Normalization

The normalization of Satallax [.] consists of the following parts:

Standardisation

When given a THF file Satallax first preprocesses the problem to remove all logical constants except $\bot, \rightarrow, \forall_\sigma, =_\sigma$ and ε_σ. For convenience we continue to write $\neg s$ for $s \rightarrow \bot$. As Satallax works with classical, extensional logic, we can replace the remaining constants using equivalences like $s \vee t \equiv \neg s \rightarrow t$, $\exists x.s \equiv \neg\forall x.\neg s$ and $s \leftrightarrow t \equiv s =_o t$. This is done to reduce the complexity of the tableau calculus Satallax works with. This standardisation is only applied once, as the rules of the calculus do not introduce new logical constants that are not supported.

Reductions

Besides the basic $\beta\eta$-reduction Satallax also applies δ-reduction and removes double negations. For the former Satallax uses the equivalence $\neg\neg s \equiv s$ not just on the top-level of a formula, but on each subterm to remove double negations at any position in the formula. Besides axioms and a conjecture Satallax allows *definitions* in the input. However any definition is immediately δ-normalized away by inserting them in the assumptions. These reductions are applied to every term during Satallax search to convert them to their normal form.

Leibniz Equality

Under certain flag settings Satallax also rewrites Leibniz equality into primitive equality before starting the search. That is, it replaces terms like $\forall p.p\ s \rightarrow p\ t$ by $s = t$. Besides Leibniz equality there are many equivalent ways to express equality in higher-order logic. Satallax also regards $\forall r.(\forall x.r\ x\ x) \rightarrow r\ s\ t$ (See X5303 in [3]) and both contrapositions $\forall p.\neg p\ s \rightarrow \neg p\ t$ and $\forall r.\neg r\ s\ t \rightarrow \neg(\forall x.r\ x\ x)$ as equality.

Symmetry

Yet another kind of normalization is the way Satallax handles symmetry of equations. Satallax regards two formulae of the form $s = t$ and $t = s$ as the same literal. To achieve this it takes the formula it encounters first as default and normalizes every following appearance to this default formula.

4.3.2 Rewrites

Algorithm 6: rewrite(**Term** t, **Variable** z)

```
Input: $z \notin FV(t) \wedge z \notin BV(t)$
Output: (Term $context$, Term $replaced$, Term $inserted$) , where
        $p := \lambda z.context$ is a predicate such that $p\ replaced =_\beta t$
switch $t$ do
    case $\lambda x.p\ x$ and $x \notin FV(p)$
        return ($z\ p, \lambda f \lambda x.f\ x, \lambda f.f$)                /* eta */
    case $\neg\neg s$
        return ($z\ s, \lambda x.\neg\neg x, \lambda x.x$)
    case $\top$
        return ($z, \top, \neg\bot$)
    case $\vee$
        return ($z, \vee, \lambda x \lambda y.\neg x \to y$)
    case $\wedge$
        return ($z, \wedge, \lambda x \lambda y.\neg(x \to \neg y)$)
    case $\leftrightarrow$
        return ($z, \leftrightarrow, \lambda x \lambda y.x = y$)
    case $\exists_\sigma$
        return ($z, \exists_\sigma, \lambda p.\neg\forall_\sigma(\lambda x.\neg p\ x)$)
    case $\lambda x.s$
        $(con, re, in) \leftarrow rewrite(s)$
        return ($\lambda x.con, re, in$)
    case $s\ t$
        try
        $(con, re, in) \leftarrow rewrite(s)$
        return ($con\ t, re, in$)
        catch No_rewrite
        $(con, re, in) \leftarrow rewrite(t)$
        return ($s\ con, re, in$)
    otherwise
        throw No_rewrite
```

Even though it holds for the normalization that $[s] = s$ for all terms s, Coq does not support all the necessary equations and reductions by default. To solve this we will explicitly state every normalization step except β- and most δ-reductions in the refutation as a rewrite step.

Those steps use the Leibniz property of equality, i.e., for every predicate p and terms s and t we can infer $p\ t$ from $s = t$ and $p\ s$.

$$\forall p. p\ s \to s = t \to p\ t$$

For example, assume we want to simplify the formula $s \wedge t$ ($\equiv$ *and* $s\ t$) by expressing the conjunction with implication and False. s and t are hence fixed as *and* and $\lambda x \lambda y. \neg(x \to \neg y)$. However we still have to find a predicate p such that (p *and*) β-normalizes to $s \wedge t$. For this we use Algorithm 6. In our example p is the term $\lambda q. q\ s\ t$. We then use Leibniz with $(\lambda q. q\ s\ t)$*and* and the equality *and* $= \lambda x \lambda y. \neg(x \to \neg y)$ to get $(\lambda q. q\ s\ t)(\lambda x \lambda y. \neg(x \to \neg y))$, which reduces to $\neg(s \to \neg t)$.

4.3.3 Lazy Rewriting

Although the translation is correct if we completely normalize each term using rewrites, this can become problematic in regard of the output size. Assume we have to refute a problem with very long formulae. As rewriting requires an explicitly stated predicate practically the size of the formula, it can have a large effect on the length of the final proof script. In some test instances the script reached a size of over six megabytes, where these long formulae made up almost the complete file (see Section 5.4).

Then again, tactics that only need formulae on the branch are short because the branch will be in Coq's state and can be referenced by identifiers. Therefore we would like to avoid rewrites as much as possible.

To achieve this the translation only applies single normalization steps using rewrites if it is necessary.

4.3.4 Alternative Tableau Rules

If we look at a formula like $s \vee t$, we can ask ourselves why we should rewrite it to $\neg s \to t$ and then apply $\mathcal{T}_{\to}$ instead of just $\mathcal{T}_{\vee}$ without the rewrite. Comparing those two rules reveals that their principal formulae and side formulae are equal up to normalization. The same holds true for all logical constant that are removed by standardisation in Section 4.3.1.

Therefore we go back from the restricted tableau calculus $\mathcal{T}$ (Figure 2.2) Satallax uses to the full tableau $\mathcal{T}^{+}$ (Figure 2.3). During the translation we will then change refutation steps to a corresponding rule if this avoids rewriting the principal formulae and the new alternatives are equal to the old up to normalization.

Additionally, we can remove double negations in front of a formula by applying $\mathcal{T}_{\neg\neg}$ instead of a rewrite.

4.3.5 δ-reduction

When handling a problem which is described using many definitions, normalized formulae in the refutation can become very long. As mentioned in Section 4.3.3, this can become problematic. Therefore, we delay δ-reductions and handle them in a similar way to rewrites. To shorten the printed length of formulae definitions will only be inserted if it is necessary for the translation to continue.

As Coq will automatically try to δ-reduce if it cannot apply a tactic to a given formula, we do not have to state the reductions explicitly except for $\mathcal{T}_{MAT}$ and $\mathcal{T}_{DEC}$.

Assume we have a definition $binomial\_3\ a\ b := add\ (mul\ a\ a)\ (mul\ b\ b)$ and the formula $binomial\_3\ x_1\ y_1 \neq binomial\_3\ x_2\ y_2$. If the refutation asks for the application of $\mathcal{T}_{DEC}$, Satallax and Coq will handle this differently. While Satallax has normalized the formula and expects $\{mul\ x_1\ x_1 \neq mul\ x_2\ x_2\}$ and $\{mul\ y_1\ y_1 \neq mul\ y_2\ y_2\}$ as the alternatives, Coq could immediately apply the rule and produce the alternatives $\{x_1 \neq x_2\}$ and $\{y_1 \neq y_2\}$ instead. This would then create a discrepancy between Coq's state and the expected state and the refutation would fail.

Our solution is to put an `unfold` tactic in the proof script in such a case to ensure that Coq reaches the same result as is expected in the refutation.

4.3.6 Symmetry

The Satallax way of handling symmetry of equality makes it hard to deal with, because it is difficult to tell which form is the default. However, as Satallax only considers symmetry at the top of a formula, we can again avoid this usually by adjusting the affected refutation steps $\mathcal{T}_{MAT}$, $\mathcal{T}_{DEC}$, $\mathcal{T}_{CON}$, $\mathcal{T}_{BE}$, $\mathcal{T}_{BQ}$, $\mathcal{T}_{FE}$ and $\mathcal{T}_{FQ}$ accordingly.

For example, let us assume one step of our refutation uses $\mathcal{T}_{BE}$ on the formula $s \neq_o t$ to get the alternatives $\{s, \neg t\}$ and $\{\neg s, t\}$, but only $t \neq_o s$ is on the branch. To strictly apply the step we would have to use symmetry to rewrite it first. However, we can just apply the rule on the formula we have to get instead the slightly different alternatives $\{t, \neg s\}$ and $\{\neg t, s\}$. We now just have to switch the subrefutations to fix it again.

In a second example, let us assume we are trying to use $\mathcal{T}_{DEC}$ on the formula $p\ s \neq p\ t$ to get the alternative $\{s \neq t\}$, but only $p\ t \neq p\ s$ is on the branch. We again just apply the rule to get instead the alternative $\{t \neq s\}$. This time we have moved the symmetry down in the refutation.

Eventually, we might have to apply symmetry only in a leaf to close the branch.

4.4 Output

The last part of the implementation prints the Coq proof script. The file contains a list of tactics describing step by step the proof of the given theorem to Coq. If Coq successfully checks the script, it will have constructed a correct proof term and thereby verify our proof.

4.4.1 Coq Proof Script

When Coq starts checking the proof, it will have the axioms as hypotheses and the conjecture t as the goal. In our first tactic we replace the goal with $\bot$. By applying the double negation law $\forall x : o.\neg\neg x \to x$ we get the new goal $\neg\neg t$. As $\neg$ is just a notation, this is the same as $\neg t \to \bot$. After introducing $\neg t$ as a new hypothesis, we are left with $\bot$ as our goal. If there was no conjecture the claim $\bot$ becomes the goal. Now, the hypotheses in global and local context correspond to our branch and proving $\bot$ means closing the branch. From here we continue the refutation with our own tactics, which simulate the tableau steps.

4.4.2 Tactics

Coq allows us to define tactic macros by using tacticals to combine tactics. We use $s; t$, which applies tactic t to every subgoal created by s, $s; [t_1 \mid \ldots \mid t_m]$, which applies tactic t_n to the nth subgoal created by s, and $s \mid\mid t$, which will apply tactic t, if s fails.

Tactic macros can also have variables as arguments. Those variables simply hold strings which are inserted into the macro when it is applied. In general each macro will only need the identifiers of the necessary hypotheses and fresh identifiers for the side formulae.

Closing Tactics

```
Ltac tab_false H := apply H.
Ltac tab_conflict H H' := apply (H' H) || apply (H' (sym_eq H)).
Ltac tab_refl H := apply (H (refl_equal _)).
```

Figure 4.5: The three closing tactics.

There are three closing tactics to simulate closing a branch by proving the goal $\bot$. The first – `tab_false H` – is used when $\bot$ is an assumption and just applies this assumption to prove the subgoal.

The second tactic – `tab_conflict H H'` – uses a conflict to close the branch by first applying `H'` and then `H`, if the assumption `H'` is the negation of `H`. As mentioned in Section 4.3.6 we also consider symmetry of equality in this macro. For example, if `H'` is $s \neq t$ and `H` is $t = s$, our first attempt will fail and we instead first use the predefined name `sym_eq` on `H` to get $s = t$.

The third tactic – `tab_refl H` – proves $\perp$ from an assumption of the form $s \neq s$. We apply `H` and prove the new subgoal $s = s$ by induction using the fact that Coq is able to deduce s from `H`.

Branching and Non-branching Tactics

The remaining tactics simulate tableau steps with at least one alternative. Most of them work the same:

Assume we have a tableau step with the principal formulae $s_1, ..., s_l$ and the alternatives $\{t_{1,1}, ..., t_{1,m}\}, ..., \{t_{n,1}, ..., t_{n,m}\}$. The corresponding tactic refines the lemma

$$\begin{aligned} s_1 \to ... \to s_l \to \quad & (t_{1,1} \to ... \to t_{1,m} \to \perp) \quad \to ... \\ ... \to \quad & (t_{n,1} \to ... \to t_{n,m} \to \perp) \quad \to \perp. \end{aligned}$$

By providing the principal formulae we are left with proving n new subgoals, where we can introduce m new assumptions leaving $\perp$ again as the goal. Note that in our tableau calculus every rule has in each alternative the same number of side formulae – either one or two –, which allows us to have a uniform m. For each tableau rule we prove the corresponding lemma in advance. A few examples are given in Figure 4.6.

Exceptions are the $\forall$-, $\exists$-, cut- and choice-tactic, where we also have to give the instantiation term, witness, cut-formula or predicate respectively.

For the implementation of the mating and decomposition rule, which can have an arbitrary number of alternatives, a recursive macro is required (Figure 4.7). After an initial part both tactics call a recursive macro. Thereby, the mating tactic combines the two predicates into an inequality, which is then handled by the decomposition tactic. With each recursive call the last arguments are removed and two new subgoals are created: For the first the reduced inequality replaces the former assumption and for the second we get the inequality of the two arguments as a new assumption. The macro is then called again on the first subgoal and if the inequality cannot be reduced any further, it will be of the form $s \neq s$ and the branch can be closed.

`Lemma TImp :` $\forall s\ \ t : o.(s \to t) \to (\neg s \to \bot) \to (t \to \bot) \to \bot$.
`Ltac tab_imp H H1 := apply (TImp H) ; intros H1.`

`Lemma TAll :` $\forall (\sigma : Type)(p : \sigma o).(\forall x : \sigma.p\ \ x) \to \forall y : \sigma.(p\ \ y \to \bot) \to \bot$.
`Ltac tab_all H y H1 := apply (TAll H y) ; intros H1.`

`Lemma TNegAll :` $\forall (\sigma : Type)(p : \sigma o)$. $\neg(\forall x : \sigma, px) \to$
$(\forall y : \sigma.\neg py \to \bot) \to \bot$.
`Ltac tab_negall H y H1 := apply (TNegAll H); intros y H1.`

`Lemma TCON :` $\forall (\sigma : Type)(s\ \ t\ \ u\ \ v : \sigma).(s = t) \to (u \neq v) \to$
$(s \neq u \to t \neq u \to \bot) \to (s \neq v \to t \neq v \to \bot) \to \bot$.
`Ltac tab_con H1 H2 R1 R2 := apply (TCON H1 H2) ; intros R1 R2.`

`Lemma TBE :` $\forall s\ \ t : o.(s \neq t) \to (s \to \neg t \to \bot) \to (\neg s \to t \to \bot) \to \bot$.
`Ltac tab_be H H1 H2 := (apply (TBE H) ; intros H1 H2).`

`Lemma TFE :`$\forall (\sigma\ \ \tau : Type)(s\ \ t : \sigma\tau)$. $(s \neq t) \to$
$(\neg(\forall x : \sigma.s\ \ x = t\ \ x) \to \bot) \to \bot$.
`Ltac tab_fe H H1 := (apply (TFE H) ; intros H1).`

Figure 4.6: Examples for typical branching tactics: $\mathcal{T}_{\to}$, $\mathcal{T}_{\forall}$, $\mathcal{T}_{\neg\forall}$, $\mathcal{T}_{CON}$, $\mathcal{T}_{BE}$ and $\mathcal{T}_{FE}$. Note that $\mathcal{T}_{\neg\forall}$,$\mathcal{T}_{BE}$ and $\mathcal{T}_{FE}$ require classical, propositional extensionality and functional extensionality .

Simple Types

As described in Section 3.2 we allow variables that are not blocked to be introduced not only by an ∃-step, but also by ∀- and cut-steps. However, we cannot just introduce a new variable in Coq as this would ignore the possibility that their type is empty. Therefore we define simple types as a construct in Coq which combines a Coq base type with a proof of its inhabitation. Additionally we give an arrow operator for creating function types from simple types.

We use this in the `tab_inh` tactic to introduce fresh variables before applying cut or ∀-steps.

```
Lemma TMat :  ∀(σ : Type)(s  t : σ)(p  q : σo). p  s → ¬q  t →
(p ≠ q → ⊥) → (s ≠ t → ⊥) → ⊥.

Lemma TDec :  ∀(σ  τ : Type)(s  t : σ)(p  q : στ). p  s ≠ q  t →
(p ≠ q → ⊥) → (s ≠ t → ⊥) → ⊥.

Ltac tab_dec' R := (refine (TDec R _ _); clear R;
[intro R; tab_dec' R | intro R]) || (apply R; reflexivity).

Ltac tab_dec H R := (apply H; reflexivity) ||
(refine (TDec H _ _);[intro R; tab_dec' R | intro R] ).

Ltac tab_mat H1 H2 R := refine (TMat H1 H2 _ _);
[(intro R; tab_dec' R ) | intro R]) ||
(refine (TMat H2 H1 _ _);[(intro R; tab_dec' R ) | intro R].
```

Figure 4.7: The Mating and Decomposition tactics.

Choice

Similar to Leibniz-equality, there are several ways to describe a choice operator in higher-order logic. If Satallax detects a hypothesis of one of the following two forms:

$$\forall (q : \sigma o), (\exists y : \sigma, q\ y) \to q(\varepsilon\ q)$$

$$\forall (q : \sigma o)(y : \sigma). q\ y \to q(\varepsilon\ q)$$

for some variable ε of type $(\sigma o)\sigma$, it will forget this hypothesis and instead consider ε as a choice operator in the search.

Therefore, we need a tactic macro for both cases: One, where we have a choice operator, and another, where we have a name with equivalent properties provided by an assumption. In both cases the tactic begins with a cut on $\forall x, \neg px$, where p is the predicate we want to use choice on.

On the first subgoal we can introduce our new assumption and have $\bot$ again as the goal. On the second subgoal we use a corresponding lemma. In the first case we use the property of the choice operator and in the second we consecutively try lemmas for each higher-order variation until one of them matches the given hypothesis. The lemmas and macros are shown in Figure 4.8.

`Lemma TSeps` $:\forall(\sigma : SType)(p : \sigma o).(p(Sepsilon\ \ p) \to \bot) \to \forall x : \sigma.\neg p\ \ x$.

`Lemma TSeps'` $:\forall(\sigma : SType)(\varepsilon : (\sigma o)\sigma)(p : \sigma o).$
$(\forall(q : \sigma o)(y : \sigma).q\ \ y \to q(\varepsilon\ \ q)) \to (p(\varepsilon\ \ p) \to \bot) \to \forall x : \sigma.\neg p\ \ x$.

`Lemma TSeps''` $:\forall(\sigma : SType)(\varepsilon : (\sigma o)\sigma)(p : \sigma o).$
$(\forall q : \sigma o.\neg(\forall y : \sigma.\neg q\ \ y) \to q(\varepsilon\ \ q)) \to (p(\varepsilon\ \ p) \to \bot) \to \forall x : \sigma.\neg p\ \ x$.

`Lemma TSeps'''` $:\forall(\sigma : SType)(\varepsilon : (\sigma o)\sigma)(p : \sigma o).$
$(\forall q : \sigma o, (\exists y : \sigma.qy) \to q(\varepsilon\ \ q)) \to (p(\varepsilon\ \ p) \to \bot) \to \forall x.\neg p\ \ x$.

```
Ltac tab_choice p R := (cut (∀x.¬p x);
[intro R | (refine (TSeps p _); intro R) ]).
Ltac tab_choice' p H R := (cut (∀x.¬p x);
[intro R | ((refine (TSeps' H _) || refine (TSeps'' H _) ||
refine (TSeps''' H _)); intro R)]).
```

Figure 4.8: The choice tactic.

Rewrites

Although Coq already has a rewrite tactic, we again define our own macros for rewriting, because Coq's tactic does not provide the desired level of control. Instead we use another of Coq's predefined names:

$$eq\_ind : \forall\{\sigma : Type\}(x : \sigma)(P : \sigma o).P\ x \to \forall y : \sigma.x = y \to P\ y$$

Here we use the predicate P we have extracted in the translation (Algorithm 6). For each kind of rewrite we define a corresponding macro, where x and y are fixed and the equality $x = y$ has again been proven beforehand. Thus each tactic only needs the predicate P and the identifier of $P\ x$. The result $P\ y$ is then β-normalized and introduced as a new hypothesis (Figure 4.9).

```
Lemma eq_or_imp :   $or = \lambda x\ y : o.\neg x \to y$.

Ltac tab_rew_or H R con :=
generalize (eq_ind $or$ (con) H ($\lambda (xy\ :\ o).\neg x\ \to\ y$) eq_or_imp);
intros R; simpl in R.

Lemma eq_eta :$\forall \sigma\ \tau : Type.(\lambda f : \sigma\tau.\lambda x : \sigma.f\ \ x) = (\lambda f : \sigma\tau.f)$.

Ltac tab_rew_eta H R con :=
generalize (eq_ind ($\lambda f\ \ x.f\ \ x$) (con) H ($\lambda f.f$) eq_eta);
intros R; simpl in R.

Lemma eq_leib :$\forall \sigma : Type.(\lambda s\ \ t : \sigma.\forall p : \sigma o.p\ \ s \to p\ \ t) = (\lambda s\ \ t : \sigma.s = t)$.

Ltac tab_rew_leib H R con := generalize
(eq_ind ($\lambda s\ \ t.\forall p : \_o.p\ \ s \to p\ \ t$) (con) H ($\lambda s\ \ t.s = t$) eq_leib);
intros R; simpl in R.
```

Figure 4.9: Examples for rewrite tactics.

5 Results and Examples

In this chapter we want to give some examples from the TPTP [22] and the results of the implementation under certain flag settings.

5.1 "Easy" and "Hard" Problems

In Section 6 of [10] some of the success of Satallax is attributed to its ability to solve problems by brute force. One such problem is SYO181ˆ5 which is known as McCarthy's Tough Nut or the mutilated checkerboard problem [4, 20]. The problem says that it is impossible to cover each tile of a checkerboard with two opposing corners removed with dominoes. Satallax is able to almost immediately solve the problem formulated in propositional logic. It thereby creates just 1154 clauses with 684 clauses required to show unsatisfiability. We could therefore describe the problem as "easy" for Satallax. Creating a refutation however does not succeed. Even after searching for several minutes and creating a partial refutation with several ten thousand steps, the search does not finish. Thus, while easy for Satallax the problem is "hard" for our implementation.

The other extreme is the second problem described in [10] that is not in TPTP:

$$(\forall x.\forall y.f\,x = f\,y) \to \exists g.\forall x(g(f\,x)) = x$$

Here, it takes Satallax about a minute to produce over one hundred thousand clauses, while trying higher-order instantiations. However, with only ten of them ultimately needed creating a proof script is done in an instant. This time creating the proof is "easy" as Satallax has done the "hard" work by finding the right instantiation term.

Fortunately, the higher-order case is more interesting for constructing a proof because there are already many provers that can handle the propositional case well.

5.2 Effect of Adding Symmetric Steps

In Section 4.2.1 we described that putting certain clauses back after the reduction to the core can make the resulting refutation shorter. We observed this first on the SYO500^1.00* problems. Those problems are variants of Kaminski's equation [18]. For example $f_1(f_1(f_1(f_2\,x))) = f_1(f_2(f_2(f_2\,x)))$ with x of type o and f_1, f_2 of type oo. In Figure 5.1 we give the results for these problems for two flag settings. In the first we only use the steps encoded by the core, while in the second setting we add for each mating rule another one that has the signs of the principal formulae switched. For example, if we have the steps $\langle A, \{f\,x, \neg f(f(f\,x))\}, \{x \neq f(f\,x)\}\rangle$ we will also allow $\langle A, \{f(f(f\,x)), \neg f\,x\}, \{f(f\,x) \neq x\}\rangle$ (see Figure 5.2). While we repeatedly run out of enabled steps in the first setting and have to resort to cut, after adding more steps we get refutations entirely without cuts which are on average 5% shorter.

Problem	UNSAT core	W/o adding steps		With adding steps	
	clauses	steps	cuts	steps	cuts
SYO500^1	15	21	0	21	0
SYO500^1.002	31	73	2	69	0
SYO500^1.003	47	195	6	195	0
SYO500^1.004	63	377	10	357	0
SYO500^1.005	79	773	16	741	0
SYO500^1.006	95	1597	44	1509	0
SYO500^1.007	111	3221	88	3045	0
SYO500^1.008	127	6493	188	6117	0

Figure 5.1: The results for the Kaminski problems: Number of clauses in the core, number of steps and cuts in the refutation without adding clauses and with adding steps.

5.3 Effect of Semantic Branching

In Section 4.2.4 we describe semantic branching as a way to use cut to reduce the size of a refutation. An example for this are the four problems SYO068^4.001, SYO068^4.005, SYO068^4.010 and SYO068^4.020 which are modal logic problems from the ILTP [24] embedded in THF. In the problems we have a reflexive and transitive relation r and $n+1$ predicates p_0 to p_n.

We have $\forall x.p_0\,x$ as the conjecture and as assumptions $\forall x.p_n\,x$ and

$$\forall z.(\forall x.r\,z\,x \to p_i\,x) \to \forall y.r\,z\,y \to (\forall x.r\,y\,x \to p_i\,x) \to \forall x.r\,y\,x \to p_{i-1}\,x$$

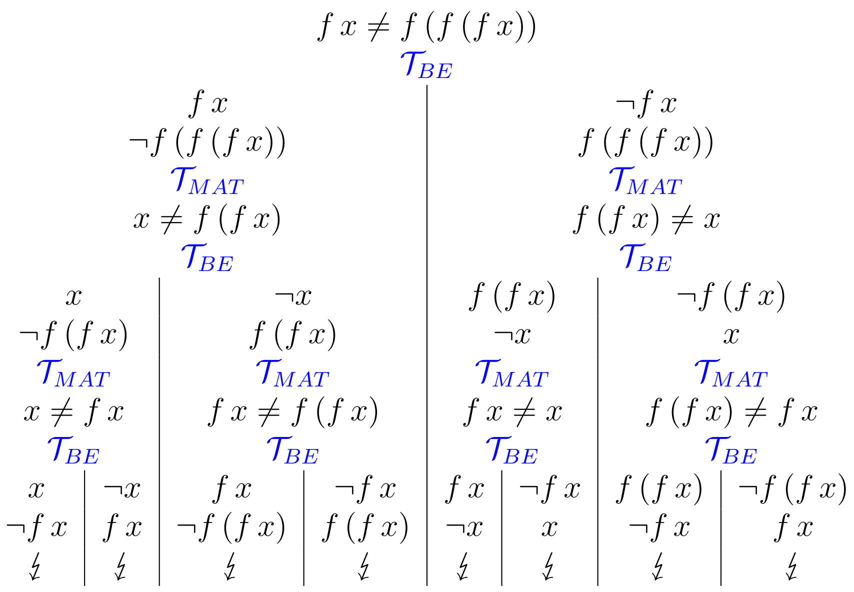
$f\,x \neq f\,(f\,(f\,x))$
$\mathcal{T}_{BE}$
$f\,x$, $\neg f\,(f\,(f\,x))$, $\mathcal{T}_{MAT}$, $x \neq f\,(f\,x)$, $\mathcal{T}_{BE}$
$\neg f\,x$, $f\,(f\,(f\,x))$, $\mathcal{T}_{MAT}$, $f\,(f\,x) \neq x$, $\mathcal{T}_{BE}$
x, $\neg f\,(f\,x)$, $\mathcal{T}_{MAT}$, $x \neq f\,x$, $\mathcal{T}_{BE}$
$\neg x$, $f\,(f\,x)$, $\mathcal{T}_{MAT}$, $f\,x \neq f\,(f\,x)$, $\mathcal{T}_{BE}$
$f\,(f\,x)$, $\neg x$, $\mathcal{T}_{MAT}$, $f\,x \neq x$, $\mathcal{T}_{BE}$
$\neg f\,(f\,x)$, x, $\mathcal{T}_{MAT}$, $f\,(f\,x) \neq f\,x$, $\mathcal{T}_{BE}$
x, $\neg f\,x$, $\lightning$ | $\neg x$, $f\,x$, $\lightning$ | $f\,x$, $\neg f\,(f\,x)$, $\lightning$ | $\neg f\,x$, $f\,(f\,x)$, $\lightning$ | $f\,x$, $\neg x$, $\lightning$ | $\neg f\,x$, x, $\lightning$ | $f\,(f\,x)$, $\neg f\,x$, $\lightning$ | $\neg f\,(f\,x)$, $f\,x$, $\lightning$

Figure 5.2: The refutation of Kaminski's equation SYO500^1. As the application of the corresponding mating steps is followed by refutations that seem symmetric, we call the steps symmetric.

for every $i \in 1, ..., n$. Table 5.3 shows the results for these problems. The number of steps in the refutation grows exponentially with the number of clauses in the UNSAT core if we just use syntactic branching. However if we apply semantic branching on implication the size of the refutations becomes linear. We give the refutation with semantic branching for SYO068^4.001 in Figure 5.4. There we can see that the refutation for one side of the cut is only done once. Without the cut it appears twice on the other side.

Problem	UNSAT core	syntactic branching		semantic branching	
	clauses	steps	cuts	steps	cuts
SYO068^4.001	16	20	0	19	1
SYO068^4.005	39	324	0	59	5
SYO068^4.010	69	10249	0	109	10
SYO068^4.020	129	1144300	0	209	20

Figure 5.3: The results for the four SYO068^4 problems: Number of clauses in the core, number of steps in the refutation without and with semantic branching.

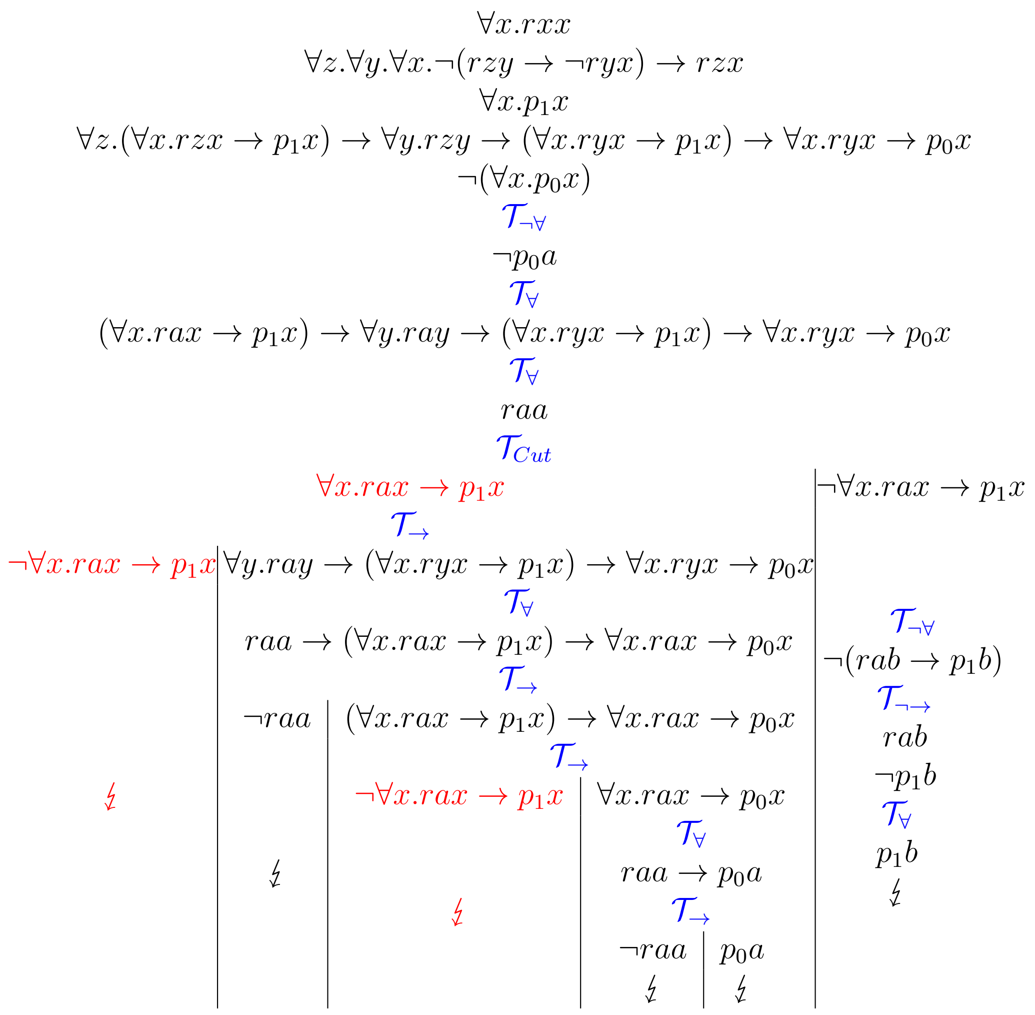
$\forall x.rxx$
$\forall z.\forall y.\forall x.\neg(rzy \to \neg ryx) \to rzx$
$\forall x.p_1x$
$\forall z.(\forall x.rzx \to p_1x) \to \forall y.rzy \to (\forall x.ryx \to p_1x) \to \forall x.ryx \to p_0x$
$\neg(\forall x.p_0x)$
$\mathcal{T}_{\neg\forall}$
$\neg p_0a$
$\mathcal{T}_{\forall}$
$(\forall x.rax \to p_1x) \to \forall y.ray \to (\forall x.ryx \to p_1x) \to \forall x.ryx \to p_0x$
$\mathcal{T}_{\forall}$
raa
$\mathcal{T}_{Cut}$
$\forall x.rax \to p_1x$
$\mathcal{T}_{\to}$
$\neg\forall x.rax \to p_1x$
$\lightning$
$\forall y.ray \to (\forall x.ryx \to p_1x) \to \forall x.ryx \to p_0x$
$\mathcal{T}_{\forall}$
$raa \to (\forall x.rax \to p_1x) \to \forall x.rax \to p_0x$
$\mathcal{T}_{\to}$
$\neg raa$
$\lightning$
$(\forall x.rax \to p_1x) \to \forall x.rax \to p_0x$
$\mathcal{T}_{\to}$
$\neg\forall x.rax \to p_1x$
$\lightning$
$\forall x.rax \to p_0x$
$\mathcal{T}_{\forall}$
$raa \to p_0a$
$\mathcal{T}_{\to}$
$\neg raa$
$\lightning$
p_0a
$\lightning$
$\neg\forall x.rax \to p_1x$
$\mathcal{T}_{\neg\forall}$
$\neg(rab \to p_1b)$
$\mathcal{T}_{\neg\to}$
rab
$\neg p_1b$
$\mathcal{T}_{\forall}$
p_1b
$\lightning$

Figure 5.4: Refutation for problem SYO068$\hat{4}$.001 with semantic branching. Due to the cut on $\forall x.r\,a\,x \to p_1\,x$ the subrefutation on the last branch appears only once instead of twice on the first and third branch.

5.4 Effect of Lazy Rewriting

In Section 4.3.3 we explained why we apply lazy rewriting. We call the opposite eager rewriting, i.e., we use rewrites to get every term in a form Satallax regards as normal. With eager rewriting 30 problems from set theory among the 261 in the SEU domain of the TPTP that Satallax can solve in MODE1 result in proof scripts with a size over one megabyte. However if we avoid rewrites whenever possible, the proof script's size is below 100 kilobytes. In Figure 5.5 we can see its effect on the five largest results, where lazy rewriting even avoids rewriting completely.

Problem	UNSAT core	eager rewrite			lazy rewrite		
	clauses	steps	rew.	script	steps	rew.	script
SEU808^1	326	940	615	4578.5	325	0	62.4
SEU809^1	327	941	615	4583.2	326	0	62.6
SEU812^1	332	946	615	4600.0	331	0	63.2
SEU818^1	339	1027	689	6663.5	338	0	64.9
SEU821^1	344	1069	726	8240.4	343	0	65.3

Figure 5.5: The results for several SEU problems: Number of clauses in the core, number of steps, rewrites and script size in kilobytes of the refutation with eager and lazy rewrite.

6 Conclusion

In this thesis, we have given a procedure to create a tableau refutation from the result of Satallax. We can extract a finite tableau calculus from the propositionally unsatisfiable set of clauses in the final state of Satallax. This finite calculus has to be extended by analytic cut and due to the pre-selected existential witnesses variables cannot be freely introduced by rules other than the $\exists$-rule. We have given a formal proof that we can refute the initial branch in this restricted tableau. We further noticed that the proof still holds for a minimal unsatisfiable core, which again drastically reduces our tableau calculus.

Based on these results we implemented our three-phased method as a post-processing step of Satallax. We first search for a simple refutation in the restricted tableau determined by the main search of Satallax. Then we complete the refutation by taking care of normalizations, while we carefully avoid explicit rewriting to prevent blowing up the size of the proof. In the last phase we output the refutation as a Coq proof script, where we have prepared lemmas and tactic macros to encode the refutation steps.

6.1 Future Work

Although the implementation successfully creates a Coq proof script for many cases, there are still cases we cannot handle yet or where we could improve the algorithm. However, we are certainly not short of ideas.

6.1.1 Dynamic Computation of UNSAT-Cores

In the current implementation we use Picomus at the beginning to reduce the clauses in the final state of Satallax to an unsatisfiable core. However, this set is only a core at this point in the search. After the first branching we can already not tell which clauses are necessary on either branch. Therefore we could continue to call Picomus during the search to keep the set of clauses minimal.

In the best case this might equally split the remaining clauses among the branches. We could even use this as a heuristic by choosing the next step that comes closest to this ideal. The main drawback though would be the cost of calling PICOMUS. An acceptable compromise could be to call it only at nodes close to the root or while the number of clauses is above a certain threshold.

6.1.2 Learning

In Section 4.2.3 about tracking dependencies we already noted the similarity between conditions and learnt clauses. While it is probably not possible to prevent repeated subrefutations, we could at least avoid wasting time on creating them twice or more. If we learn a refutation's condition and encounter later a branch that contains the condition, we will be able to just copy the refutation we already found. Furthermore, we can avoid printing and checking it more than once in the proof script if we include it as a lemma into the script and later refer to the lemma when used in the refutation. This would require building a suitable data structure that allows to look up quickly whether an element is a subset of the current branch while keeping the space requirements moderate.

6.1.3 Pattern Clauses

An important part of Satallax are pattern clauses, which contribute to about 30% of the solutions Satallax finds [10]. Unfortunately, our algorithm does not include them at this point.

6.1.4 Alternative Approach

We already justified why our implementation does not follow the algorithm the proof in Section 3.3 describes. However, it can be an alternative for problems that are too large for our algorithm.

In a first phase we have to remove $\exists$-clauses from the clause set by satisfying them with cuts, until on all branches no variable is blocked anymore. Then we call a SAT solver on each of them, which is able to give short unsatisfiability proofs like *resolution graph* [15] or *conflict clause proofs* [16]. The returned proofs can be translated into refutations in a straightforward way.

For example, let us assume we get a resolution graph, i.e., every learnt clause is assigned a node with the clauses it is inferred from as parents. The initial clauses are source-nodes and the empty clause is assigned to a sink-node.

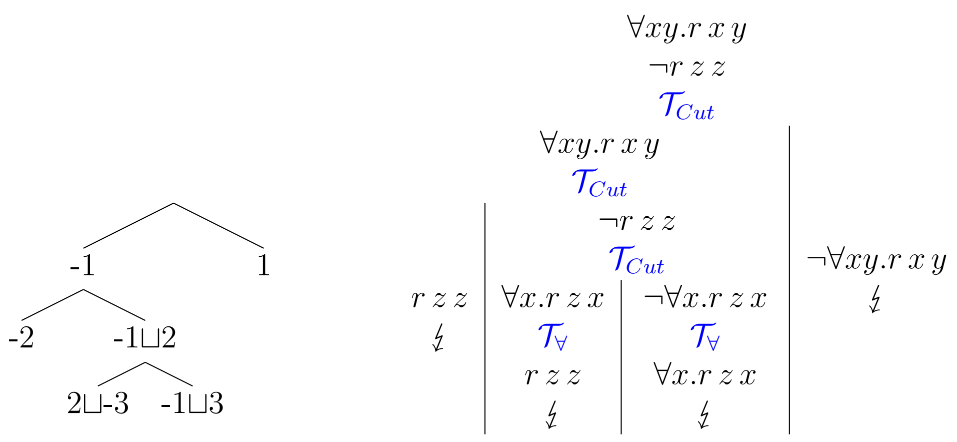

Figure 6.1: Resolution graph translated into refutation, where $1 := \lfloor \forall xy.r\ x\ y \rfloor$, $2 := \lfloor r\ z\ z \rfloor$ and $3 := \lfloor \forall x.r\ z\ x \rfloor$. $\mathfrak{C} := \{1, -2, -1{\sqcup}3, 2{\sqcup}{-3}\}$ is propositionally unsatisfiable. The graph is represented as a tree: The root is the sink with the empty clause, while the leafs are the source nodes. The parent/children relation of graph and tree are inverted.

We recursively construct the refutation beginning with the initial branch assigned to the empty clause. For each open branch we look at the parents of the assigned clause to identify the atom both have the opposite literal of and then cut on the corresponding formula. We assign the resulting branches to the parent with the opposite literal cut has added. In the end every branch is now either closed as it conflicts with an assumption or can be closed using the step encoded by the assigned clause. We give an example in Figure 6.1.

The advantage of this approach is that we do not have to search for the refutation except for the beginning, but can rely on established software instead. However, we suspect that any problem where this would outperform our implementation might be too large for Coq to check anyway. Furthermore we would have to output the formula for each cut in the proof script.

6.1.5 Satisfiability proof

The topic of this thesis has been about giving a proof for the case that the given list of axioms is refutable. However, sometimes Satallax arrives at the solution that it is not. This is the case when Satallax cannot add any more clauses to its state although the set of clauses is still propositionally satisfiable.

To verify this result differs largely with what we are doing, but is not any more complicated. As the set of clauses is propositionally satisfiable, MiniSat returns a propositional assignment Φ as its answer. If we turn Φ into a set A of higher-order formulae as we did before with our branches, A will fulfil the *evidence (Hintikka)* conditions [6, 17]. By the Model Existence Theorem there is a Henkin model satisfying A [10], which proves that our set of axioms is satisfiable.

We can give A as our proof and write a simple verifier checking the *evidence (Hintikka)* conditions on A.

Bibliography

[1] Coq website. `http://coq.inria.fr/`.

[2] Satallax. `http://www.ps.uni-saarland.de/~cebrown/satallax/`.

[3] Peter B. Andrews. Classical type theory. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 15, pages 965–1007. Elsevier Science, 2001.

[4] Peter B. Andrews and Matthew Bishop. On sets, types, fixed points, and checkerboards. In *Proceedings of the 5th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, pages 1–15, London, UK, 1996. Springer-Verlag.

[5] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Efficient generation of unsatisfiability proofs and cores in SAT. In *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR '08, pages 16–30, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Julian Backes and Chad E. Brown. Analytic tableaux for higher-order logic with choice. In Reiner Hähnle Jürgen Giesl, editor, *Automated Reasoning: 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010, Proceedings*, volume 6173 of *LNCS/LNAI*, pages 76–90. Springer, 2010.

[7] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, H. Herbelin, G. Huet, P. Manoury, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual Version 8.3.* INRIA-Rocquencourt-CNRS-ENS Lyon, June 2004.

[8] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[9] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4(2-4):75–97, 2008.

[10] Chad E. Brown. Reducing higher-order theorem proving to a sequence of SAT problems. In *CADE – the 23rd International Conference on Automated Deduction (To Appear)*. LNCS, Feb 2011. To Appear.

[11] Marcello D'Agostino. Are tableaux an improvement on truth-tables? cut-free proofs and bivalence. *Journal of Logic, Language and Information*, 1:235–252, 1992.

[12] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, July 1962.

[13] N. de Bruijn. The Mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on Automatic Demonstration*, pages 29–61. Lecture Notes in Mathematics, 125, Springer, 1970.

[14] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 333–336. Springer Berlin / Heidelberg, 2004.

[15] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *In Seventh Int'l Symposium on AI and Mathematics*, 2002.

[16] Evgueni Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *IN DESIGN, AUTOMATION AND TEST IN EUROPE (DATE'03)*, pages 886–891, 2003.

[17] K. Jaakko J. Hintikka. Form and content in quantification theory. Two papers on symbolic logic. *Acta Philosophica Fennica*, 8:7–55, 1955.

[18] Matthias Höschele. Towards a semiautomatic higher-order tableau prover, 2009. Bachelor's Thesis, Universität des Saarlandes.

[19] Joaäo P. Marques-silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[20] John McCarthy. A tough nut for proof procedures. Stanford Artificial Intelligence Project Memo No. 16, July 1964.

[21] Frank Pfenning. Analytic and non-analytic proofs. In *Proceedings of the 7th International Conference on Automated Deduction*, pages 394–413, London, UK, 1984. Springer-Verlag.

[22] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[23] Geoff Sutcliffe and Christoph Benzmüller. Automated Reasoning in Higher-Order Logic using the TPTP THF Infrastructure. *Journal of Formalized Reasoning*, 3(1):1–27, 2010.

[24] Christoph Kreitz Thomas Raths, Jens Otten. The ILTP Problem Library for Intuitionistic Logic - Release v1.1. *Journal of Automated Reasoning*, 2006. To Appear.