Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Jan Jakubův

Automated Theorem Proving Using the Tableaux Method

Supervisor: Prof. RNDr. Petr Štěpánek, DrSc.
Study program: Computer Science

First of all I would like to thank my supervisor, Petr Štěpánek, for his advice and many helpful comments.

Furthermore I would like to thank Josef Urban for offering additional heplful resources.

I declare that I have written this Master Thesis on my own and listed all used sources. I agree with lending of the Thesis.

Prague, August 11, 2006

Jan Jakubův

Contents

Název práce: Automatické dokazování vět s použitím tableaux metod
Autor: Jan Jakubův
Katedra (ústav): Katedra teoretické informatiky a metamatické logiky
Vedoucí diplomové práce: Prof. RNDr. Petr Štěpánek, DrSc.
e-mail vedoucího: petr.stepanek@mff.cuni.cz
Abstrakt: V této práci studujeme tableaux kalkul a podobné metody. Nejprve zavádíme obvyklé pojmy a poté představujeme tableaux kalkul pro logiku prvního řádu. Dále představujeme automatický dokazovač vět autorů Beckert a Possega [5]. Rozšiřujeme tento dokozovač o metodu, která mž urychlit jeho práci. Výsledný program porovnáváme s jeho původní verzí. Dále představujeme deduktivní systém T_ξ autorů Degtyarev a Voronkov [3]. Hlavním přínosem této práce je implementace tohoto systému. Dále také vyvíjíme khinovnu pro práci s objekty logiky prvního řádu v programovacím jazyce Python.
Klíčová slova: Automatické dokazování, tableaux metody, eliminace rovnosti

Title: Automated Theorem Proving Using the Tableaux Method
Author: Jan Jakubův
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: Prof. RNDr. Petr Štěpánek, DrSc.
Supervisor's e-mail address: petr.stepanek@mff.cuni.cz
Abstract: In this paper we are studying the Tableaux Calculus a related methods. We adopt basic notions and present the tableaux calculus for the First-Order Logic. Then we present the Beckert and Possega [5] prover. We extend this prover with speed-up technique and compare the results with the original prover. Then we present the tableaux T_ξ system of Degtyarev and Voronkov [3] supposed to handle the equality. The main benefit of this paper is the implementation the the T_ξ system prover. We also present a library providing tools to work with the objects of the First-Order Logic in the programming language Python.
Keywords: Automated Theorem Proving, Tableaux Method, Equality Elimination

Chapter 1

Introduction

Here we present a brief overview of definitions used in the paper.

1.1 Preliminaries

By a *first-order signature* Σ we mean a set of function and predicate symbols. X denotes set of variables. Given a signature Σ we define set $\mathcal{T}(\Sigma, X)$ of terms and set $\mathcal{A}(\Sigma, X)$ of atomic formulas in a usual way. We shall write $\mathcal{T}_0(\Sigma)$ and $\mathcal{A}_0(\Sigma)$ to denote set of ground terms (i.e. not containg variables). If Σ and X is clear from the context we shall write $\mathcal{T}$ and $\mathcal{A}$ as well as $\mathcal{T}_0$ and $\mathcal{A}_0$. For skolemization we shall use infinite set $\mathcal{F}_{sko}$ of *Skolem function symbols* that is disjoint with function symbols in Σ. A signature Σ extended by symbols from $\mathcal{F}_{sko}$ will be denoted as Σ^*.

A *literal* is either an atomic formula or negation of such that. A *clause* is a set of literals interpreted as a disjunction. An *equation* is an atomic formula of the form $s = t$ and a *disequation* is of the form $\neg(s = t)$ denoted also as $s \neq t$ where s and t are terms. Given a term t we write t_s to denote that the term t contains term s as a subterm. We shall also use the expression $t_s[s']$ to denote a term where s' is substituted for a single occurence of s in t.

A *substitution* σ is a mapping from a finite subset of variables to terms and is denoted as $\{x_1/\sigma(x_1), \ldots x_n/\sigma(x_n)\}$. We shall use symbol ε to denote the empty substitution. We write an application of substitution ϑ to the term t as $t\vartheta$. We write a composition of substitutions ϑ and σ as $\vartheta\sigma$. A substitution σ is called an *unifier* of terms t and s if $t\sigma = s\sigma$. A substitution σ is called a *subset unifier* of the clause D_1 against the clause D_2 iff $D_1\sigma \subseteq D_2\sigma$. Let $\sigma \leq \theta$ means that there is a substitution τ such that $\sigma\tau = \theta$. Then σ is the *most general unifier* (respectively the *most general subset unifier*) of s and t (respectively of D_1 against D_2) iff it is minimal w.r.t. $\leq$ among all unifiers of s and t (respectively among all subset unifiers of D_1 against D_2).

A first-order formula is called to be in *negation normal form* (NNF) if each occurence of negation symbol in it is part of a literal. There is a provability preserving translation of any first-order logic formula to the formula in NNF.

Chapter 2

The Tableaux Calculus

The Tableaux Calculus is a decision procedure used to solve the problem of satisfiability of the first-order logic formula. In this section we first adopt some basic notions used in tableaux literature. Then we present the tableau calculus with unification in the style used in Hanhle [2]. Finally we demostrate the tableau calculus on a simple example.

2.1 Basic Notions

By a *compound first-order formula* we mean a first-order formula such that it's NNF is not a literal. In Hanhle [2] compound first-order formulas are divided into four types according to the leading logic compound. Conjuctive formulas are called α, disjunctive β, universally quantified γ and existencial formulas δ. We shall use these letters to denote formula of coresponing type. To denote first and second part part of conjunction (respectively disjunction) we shall write α_1 and α_2 (respectively β_1 and β_2). If we want to denote the variable x bounded in γ- or δ-formulas by leading quantifier, we shall write $\gamma(x)$ or $\delta(x)$ respectively. The $\gamma(x)$ or $\delta(x)$ formula without it's leading quantifier is denoted as γ_1 respectively δ_1 and the result of substituting term t for x in γ_1 or δ_1 is denoted as $\gamma_1(t)$ or $\delta_1(t)$ respectively. Note that there is no type for formulas that are negations of compound formulas. Such formulas can be eliminated by the translation into the negation normal form.

2.2 Tableaux Rules

In the tableaux method described here skolemization is done during the proof search. Following *sko* function is used.

Definition. Let Σ be a first-order signature and $>$ arbitrary but fixed ordering on $\mathcal{F}_{sko}$. The function *sko* assigns to each δ-formula over Σ^* a symbol $sko_\delta \in \mathcal{F}_{sko}$ such that

1. $sko_\delta > f$ for all $f \in \mathcal{F}_{sko}$ occuring in δ and
2. for all δ-formulas δ and δ' over Σ^* the symbols sko_δ and $sko_{\delta'}$ are identical iff δ and δ' are identical up to variable renaming.

Tableaux are usually presented as trees (in this case binary) with nodes labeled by first-order formulas. To construct such a tree we shall use rules that can expand branch of the tree with some formula. The following table contains four rules schemata:

$$\begin{array}{cccc}
\dfrac{\alpha}{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}} & \dfrac{\beta}{\beta_1 \mid \beta_2} & \dfrac{\gamma(x)}{\gamma_1(y)} & \dfrac{\delta(x)}{\delta_1(sko_\delta(x_1,...,x_n))} \\
 & & \text{where } y \text{ is} & x_1,\ldots,x_n \text{ are the} \\
 & & \text{a fresh variable} & \text{free variables in } \delta
\end{array}$$

Each rule scheme corresponds to one of four compound formula types defined above. We shall denote these rules in turn α, β, γ and δ. Premises and extensions of each rule are separated by horizontal bar. Vertical bar in the β-rule separates different *extensions.* Formulas in the same extension (i.e. α_1 and α_2 in the α-rule) are implicitly conjunctively connected whereas different extensions are connected disjunctively. The α rule tells us that whenever the α-formula α is allready placed on some branch, we can expand the branch with one of α_1 or α_2. In the case of β-rule when we have the β-formula β on the branch, we can split the branch and place β_1 as the left successor of the last node of the branch and β_2 as the right one. In γ-rule the γ-formula $\gamma(x)$ is stripped from the leading universal quantifier and the branch can be extented by formula γ_1 where the fresh variable y is substituted for x. The δ-rule replaces existencially quantified variables with a Skolem term.

2.3 Tableaux with Unification

Here we present the tableaux calculus for the First-Order Logic that is in Hanhle [2] called *tableaux with unification.* The *tableau over* Σ^* is a finitely branching tree whose nodes are first-order formulas over Σ^*. A *tableaux calculus* has one initialization rule (the axiom) and two inference rules each deriving tableau from another tableau. Inferable objects of the calculi are defined in the following way.

Definition Let Φ be a set of sentences over Σ. Than the *tableau with unification for* Φ is a tableau constructed with the following rules:

1. The tree consisting of a single node *true* is a tableau for Φ (*initialization rule*).

2. Let T be a tableau for Φ, B a branch of T, and φ a formula in $B \cup \Phi$. Consider an arbitrary instance of a tableau rule scheme in the above table which has φ as a premise. Obtain the tree T' from T by expanding B with linear subtrees whose nodes are formulas in the extensions of the rule instance. Then T' is a tableau for Φ (*expansion rule*).

3. Let T be a tableau for Φ, B a branch of T, and φ and φ' literals in $B \cup \Phi$. If φ and $\overline{\varphi'}$ are unifiable with mgu σ, and T' is constructed by applying σ to all formulas in T (i.e., $T' = T\sigma$), then T' is a tableau for Σ (*closure rule*)[1]

The initialization rule constructs initial tableau consisting of one root node. The expansion rule is used to expand derived tableau. Note that the use the expansion rule with φ that is β-formula leads to branching the tableau. New tableau T shall have one more branch. Usage of expansion rule with formula φ of another type only extends the branch not increasing number of branches. The closure rule is used to close branch which means to find a unificator of two complementary literals placed on the branch and applying such substitution to the whole tableau. Our aim is to close all tableau brances.

Definition Let T be a tableau with unification for set of sentences Φ and B a branch of T. We say that B is *closed* iff $B \cup \Phi$ contains a pair φ, $\neg\varphi$ of complementary literals, otherwise, it is *open*. A tableau with unification is *closed* iff all its branches are closed.

A *tableau proof* for a set Φ of sentences over Σ is a closed tableau with unification for Φ.

Now we can finally connect the unsatisfiability of the set Σ of sentences with some derivation in tableau calculus.

Theorem (*Soundness and completness*) Let Φ be a set of sentences over Σ. Then Φ is unsatisfiable iff there is a tableau proof for Φ.

[1]Note that we use the complement $\overline{\varphi'}$ of the the literal φ' defined as $\neg\varphi'$ when φ' is an atom and as ψ' if φ' is the negation of an atom of the form $\neg\psi'$.

As a consequence of the previous theorem we can formulate following.

Conclusion Let Φ be a set of sentences over Σ and φ the sentence over Σ. Then $\Phi \vdash \varphi$ iff there is a tableau proof for $\Phi \cup \{\neg\varphi\}$.

The proof of previous theorems and more detailed description of tableaux methods can be found in Hanhle [2].

2.4 Example

As said above, to prove that $\Phi \vdash \varphi$ we want to find the tableau proof for $\Phi \cup \{\neg\varphi\}$. It can be seen as a proof by contradiction and case distinction. We should start with the negation of conjecture φ by expanding it using expansion rule as much as possible. Then we shall try to use axioms from Φ to find elementary contradiction. The branches of the constructed tableau correspond to different subcases of the proof. We shall demonstrate this in example problem of composing implications also known as Aristotelian Syllogism.

Example. Let us prove that $(\forall x)(P(x) \rightarrow Q(x))$ and $(\forall x)(Q(x) \rightarrow R(x))$ implies $(\forall x)(P(x) \rightarrow R(x))$. The previous theorem tells us to find a tableau proof for

$$\Phi = \{(\forall x)(\neg P(x) \vee Q(x)), (\forall x)(\neg Q(x) \vee R(x)), (\exists x)(P(x) \& \neg R(x))\}$$

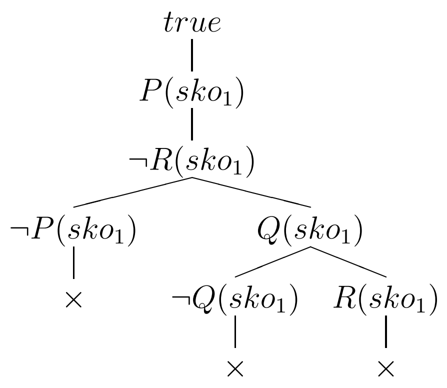
Note that we have translated all formulas into the NNF. To expand the initial tableaux we shall start with the negated conjecture first. This is usually good idea because we usually assume that our theory is consistent and thus there is no tableaux proof for the set of axioms. Hence the negated conjecture plays principal role in tableau proof. We shall apply expansion rule three times to get following tableau:

$$\begin{tikzcd}
true \arrow[d, no head] \\
(\exists x)(P(x)\&\neg R(x)) \arrow[d, no head] \\
P(sko_1)\&\neg R(sko_1) \arrow[d, no head] \\
P(sko_1) \arrow[d, no head] \\
\neg R(sko_1) \arrow[d, no head] \\
\ldots
\end{tikzcd}$$

We now have the tableau with one opened branch. We have two literals on this branch $P(sko_1)$ and $\neg R(sko_1)$. We should say that we suppose $P(sko_1)$ and $\neg R(sko_1)$ and we are trying to reach the contradiction. We shall use now, for example, the first axiom. Using expansion rule twice we will branch our tableau but we shall be able to close one branch immediately. The situation after two expansions is as follows.

$$\begin{tikzcd}
 & true \arrow[d, no head] & \\
 & P(sko_1) \arrow[d, no head] & \\
 & \neg R(sko_1) \arrow[d, no head] & \\
 & (\forall x)(\neg P(x) \vee Q(x)) \arrow[d, no head] & \\
 & \neg P(y_1) \vee Q(y_1) \arrow[dl, no head] \arrow[dr, no head] & \\
\neg P(y_1) \arrow[d, no head] & & Q(y_1) \arrow[d, no head] \\
\ldots & & \ldots
\end{tikzcd}$$

Now we can use the closure rule to close the left branch using unification $\{y_1/sko_1\}$. Recall that this unification has to be applied to the whole tableau, so that the right open branch shall after the application of closure rule end with the literal $Q(sko_1)$. Note that we have omited some irrelevant nodes of the tableau. The next step is to use the second axiom to expand the right opened tableau branch. It shall branch the opened branch but both new created branches should be closed immediately. The resulting tableau proof without irrelevant parts is in the next figure.


$true$
$P(sko_1)$
$\neg R(sko_1)$
$\neg P(sko_1)$
$Q(sko_1)$
$\times$
$\neg Q(sko_1)$
$R(sko_1)$
$\times$
$\times$

Chapter 3

The leanTAP Prover

The leanTAP is a sound and complete theorem prover for the First-Order Logic created by Berhhard Beckert and Joachim Possega Beckert and Possega [5]. leanTAP can be implemented in five lines of Prolog code. It makes the leanTAP probably the smallest theorem prover for the First-Order Logic around the world. The leanTAP is based on tableaux with unification presented in previous section. Here we present just a short overview in order to extend the leanTAP by irrelevant proof parts pruning technique and to compare it with original program. For more information about the leanTAP refer Beckert and Possega [5] or leanTAP FAQ Beckert and Possega [6].

3.1 The leanTAP Code

Here we present the core of the leanTAP code.

```
prove((A,B),UnExp,Lits,FreeV,VarLim) :- !,
    prove(A,[B|UnExp],Lits,FreeV,VarLim).

prove((A;B),UnExp,Lits,FreeV,VarLim) :- !,
    prove(A,UnExp,Lits,FreeV,VarLim),
    prove(B,UnExp,Lits,FreeV,VarLim).

prove(all(X,Fml),UnExp,Lits,FreeV,VarLim) :- !,
    \+length(FreeV,VarLim),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UnExp,[all(X,Fml)],UnExp1),
    prove(Fml1,UnExp1,Lits,[X1|FreeV],VarLim).
prove(Lit,_,[L|Lits],_,_) :-
    ((Lit= -Neg; -Lit=Neg)) ->
        (unify(Neg,L); prove(Lit,[],Lits,_,_)).

prove(Lit,[Next|UnExp],Lits,FreeV,VarLim) :-
    prove(Next,UnExp,[Lit|Lits],FreeV,VarLim).
```

In order to achieve maximal efficiency from minimal means, as much work as possible is left to the Prolog itself. First-order logic variables are represented by Prolog variables. This representation allows the leanTAP to use Prolog unification[1]. First-order function and predicate symbols are encoded using Prolog functors and first-order terms and atomic formulas are encoded using Prolog terms. To denote the negation symbol Prolog unary functor `-` is used. Conjunction and disjunction are encoded using binary functors "`,`" and "`;`". Finally Prolog term of the form `all(X,Fml)` encodes the first-order formula $(\forall x)\varphi$ where Prolog variable `X` encodes first-order variable x and Prolog term `Fml` encodes the first-order formula φ. The leanTAP also supposes skolemization to be done before translating the input formula into the Prolog term.

The leanTAP soundness and completeness is stated in the following theorem Beckert and Possega [6].

Theorem (Soundness and completness of the leanTAP)
Let Φ be the set of first-order formulas in skolemized NNF and φ the first-order formula. Let also `notfml` be the Prolog term encoding skolemized NNF form of the first-order formula $\neg\varphi$. Finally let `h` be the Prolog list of terms encoding formulas from Φ.

1. Let `d` be a natural number encoded in a Prolog term. If the query `prove(notfml,h,[],[],d)` to the program leanTAP returns *success* as an answer then $\Phi \vdash \varphi$.

2. If $\Phi \vdash \varphi$ then there is a natural number d (and corresponding Prolog term `d`) such that the query `prove(notfml,h,[],[],d)` to the leanTAP terminates with success.

When the leanTAP is runned it searches the space of all tableaux for the tableau proof. Because the tableaux calculus as presented here is *destructive*[2]

[1]For efficiency reasons most of Prolog interpreters omit occur-check during unification. On the other hand the leanTAP uses standard unification with occur-check encoded in the `unify/2` predicate. Most of Prolog interpreters have this predicate built-in with special name such as `unify_with_occurs_check/2`.

[2]see Hanhle [2] section 3.3 for definition and more information

we need to divide search space into finite parts. This is the reason of natural number d in the previous theorem. This means that For fixed d leanTAP searches for the tableau proof in the space of all tableaux constructed using at most d applications of the γ-rule. We can also choose different interpretation for d, e.g. the height of the constructed tableau. The way of dividing search space is called *completion mode* and different ways are described in Hanhle [2]. To find the right d *iterative deepening* technique is used in leanTAP . Note that the code implementing iterative deepening is not presented here.

Now we describe how the leanTAP does work in the way as short as possible. The first argument of `prove/5` predicate is representing the formula to be expanded in the current step. The second argument `UnExp` is a list of unexpaned formulas. The third argument `Lits` is a list of literals existing on the current branch. The fourth argument is a Prolog term containing variables that are free on the current branch. Finally the last argument holds the iterative deepening limit. See Beckert and Possega [5] for more explanation.

The leanTAP always tries to close the left most tableau branch first. Other branches of tableau are not represented using some argument of `prove/5`. They are represented on the Prolog call stack. According to the type of the formula to be expanded the corresponding clause of the leanTAP is called. The first clause corresponds to α-formulas, second to β and the third to γ-formulas[3]. Note that we don't have a clause for δ-formulas because we suppose the formula to be in skolemized NNF. The fourth clause is trying to close the current branch and it is always called with a literal, possibly containing variables, as a first argument. Finally the fifth clause just chooses next formula to be expanded.

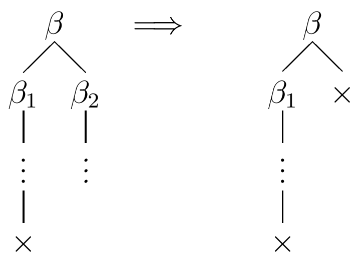
3.2 Irrelevant Proof Parts Pruning

In this section the speed-up technique of searching for the tableau proof is presented. This technique *irrelevant proof parts pruning* is in short described in Hanhle [2] section 3.5.2 but is not implemented in leanTAP . Here we introduce simple way of implementing this technique into the leanTAP . The results of tests comparing this implementation with the original leanTAP are presented in Appendix C.

[3]Note that when using γ-rule with some γ-formula, new instance of γ is added to the end of the list of unexpanded formulas. This is necessary for achieving completeness.

3.2.1 The Speed-up Technique

Now we introduce the above technique in a short way. Suppose that some branch is expanded by a β-rule. Denote in turn by β_1 and β_2 the left and right extension. Suppose that the left branch begining with the extension β_1 is closed during the proof search without using β_1. Closing the branch without using β_1 means that no literal originated from β_1 is used in any closure rule used to close any branch below β_1. Then the right branch begining with β_2 should be closed immediately. The tecnhinque is illustrated in the figure below.



3.2.2 Implementation

To implement the above method into the leanT^AP we use the concept of Prolog anonymous variables. For each formula we remeber if formula itself or some of it's subformula was used in a branch closure. For every formula this information is saved in the variable denoted `Prune`. Instead of working with formulas in the first argument, the second and the third arguments, we extend the original term of the leanT^AP `Fml` to the term `Fml-Prune`. When the formula or some of it's subformula is used in a branch closure the prune of that formula is set to the Prolog atom `true`. Let *prune of the formula* denotes value of the variable `Prune` corresponding to the formula.

When using α- or γ-rule we just propagate prunes of variables down. This way is the prune propageted to the literals. When branching tableau using the β-rule, which corresponds to the second clause of the leanT^AP , new anonymous variable is created. This anonymous variable will be the the prune of β_1 that is the left extension of the β-formula beeing expanded. Then when going to the right branch to try to close β_2, the prune of β_1 is tested. If the prune of β_1 is `true` then the leanT^AP set the prune of β to `true` and creates new anonymous variable as a prune for β_2. But if the prune of β_1 is still a variable then the current goal succeeds. When closing a branch both the prune of literal beeing added on the branch and of it's complement allready presented on the branch are unified with the Prolog atom `true`. Finally when selecting next formula to expand then the prune of the literal is stored with the literal in the third argument of the prove/5 predicate. The resulting code is presented bellow.

```
prove((A,B)-Prune,UnExp,Lits,FreeV,VarLim) :- !,
    prove(A-Prune,[B-Prune|UnExp],Lits,FreeV,VarLim).

prove((A;B)-BetaPrune,UnExp,Lits,FreeV,VarLim) :- !,
    prove(A-Beta1Prune,UnExp,Lits,FreeV,VarLim),
    nonvar(Beta1Prune) ->
        ( BetaPrune = true,
          prove(B-_,UnExp,Lits,FreeV,VarLim)
        ) ).

prove(all(X,Fml)-Prune,UnExp,Lits,FreeV,VarLim) :- !,
    \+length(FreeV,VarLim),
    copy_term((X,Fml,FreeV),(X1,Fml1,FreeV)),
    append(UnExp,[all(X,Fml)-_],UnExp1),
    prove(Fml1-Prune,UnExp1,Lits,[X1|FreeV],VarLim).

prove(Lit-Prune1,_,[L-Prune2|Lits],_,_) :-
    (Lit= -Neg; -Lit=Neg) ->
        (   ( unify(Neg,L), Prune1=true, Prune2=true ) ;
            ( prove(Lit-Prune1,[],Lits,_,_) )
        ).

prove(Lit-Prune,[Next|UnExp],Lits,FreeV,VarLim) :-
    prove(Next,UnExp,[Lit-Prune|Lits],FreeV,VarLim).
```

Chapter 4

The System T_ξ

4.1 Introduction

The system T_ξ is tableaux based deductive system presented in Degtyarev and Voronkov [3]. The main benefit of the system T_ξ is the handling of the equality. Tableaux based provers are known not to work well with the equality. The system T_ξ is based on extending a tableau prover by a bottom-up equation solver using basic superposition.

In the system T_ξ indirect version of the the tableaux calculus from previous sections is turn round to the direct one. It means that the role of logic compound will turn round. For example we shall branch the tableaux branch when expanding the branch by conjunction instead of by disjunction as in previous sections. Also the skolemization will turn round to the *anti-skolemization* which is the same as the skolemization but we just replace universally quantified formulas by their Skolem versions instead of replacing existencial quantified formulas.

4.2 Term ordering

Partial ordering is a binary transitive and irreflexive relation on some set. We say that partial ordering S on a set $\mathcal{S}$ is *total* on $\mathcal{S}$ iff for each disctinct $x, y \in \mathcal{S}$ one of (x, y) or (y, x) is in S.

Now suppose that $\succ$ is an ordering on $\mathcal{T}(\Sigma, X)$. Then $\succ$ is called *reduction ordering* iff

1. $\succ$ is well-founded and
2. if $s \succ t$ then $u_s[s\vartheta] \succ u_t[t\vartheta]$ holds for every substitution ϑ and for all terms s, t, u where s and t are subterms of u.

Reduction ordering doesn't need to be total in general, but later on we shall use reduction ordering that is total on set of ground terms $\mathcal{T}_0(\Sigma)$. We shall use the ordering $\succ_{lpo}$ called *lexicographic path ordering* (LPO[1]) that is defined as an extension of any total ordering $>_{\mathcal{F}}$ on a (finite) set of function symbols in Σ in the following way:
Let s and t denote terms and in the case that it's compound term (i.e. neither variable nor constant) let f and g denote it's leading function symbols and s_i and t_i it's i-th argument. So $s = f(s_1, \dots s_m)$ and $t = g(t_1, \dots t_n)$.
Then $s \succ_{lpo} t$ iff

1. $s_i \succeq_{lpo} t$ for some i such that $0 < i \leq m$ or
2. $f >_{\mathcal{F}} g$ and $s \succ_{lpo} t_j$ for all j that $0 < j \leq n$ or
3. $f = g$ and $s \succ_{lpo} t_j$ for all j that $0 < j \leq n$ and $(s_1, \dots, s_m) \succ_{lpo}^{lex} (t_1, \dots, t_n)$ where by $\succ_{lpo}^{lex}$ we mean lexicographical ordering on n-tuples with respect to $\succ_{lpo}$.

In the following sections we shall always suppose that $\succ$ denotes reduction ordering that is total on ground terms.

4.3 ξ-names

We say that a first-order formula is in anti-Skolem negation normal form if it consists only of literals composed using connectives $\&$, $\vee$ and by existencial quantifier $\exists$. It can be proved that it is possible to translate every first-order formula into formula in anti-Skolem negation normal form preserving it's provability in the First-Order Logic. Let ξ is a formula in anti-Skolem negation normal form. We say that φ is superformula of ψ iff ψ is a subformula of φ. Let φ is a subformula of a formula ξ. The occurrence of subformula φ of ξ is called *conjunctive* iff it is an occurrence in a subformula $\varphi \& \psi$ (or $\psi \& \varphi$). A *conjunctive superformula* of φ is a superformula of φ that is conjunctive. The *least conjunctive superformula* of φ is such conjunctive superformula ψ of φ that any other cunjunctive superformula of φ contains ψ as it's subformula.

As a simple example consider formula $P(x) \vee (Q(x) \& P(x))$. The first occurrence of $P(x)$ has no least conjunctive superformula. The least conjunctive superformula for the second occurrence of $P(x)$ is $Q(x) \& P(x)$. $Q(x) \& P(x)$ is also the least conjunctive superformula for the occurrence of $Q(x)$. We also note that every formula that has conjunctive superformula has unique least conjunctive superformula.

[1]See for example Nieuwenhuis and Rubio [12].

We now fix some enumeration $\xi_1, \ldots, \xi_n$ of all conjunctive subformulas in ξ for example by their occurrencies from the left. Let $A_1, \ldots, A_n$ are fresh predicate symbols not occurrencing in Σ. We say that $A_k(x_1, \ldots, x_m)$ is the *ξ-name of a subformula φ of ξ* iff

1. The least conjunctive superformula of φ is ξ_k and
2. $x_1, \ldots, x_m$ are all free variables occurring in ξ_k in order of their occurencies in ξ_k.

For an arbitrary subformula φ of ξ it's ξ-name need not exist. If it exists, than it is unique. The *set of ξ-names of subformula φ of ξ* we define as $\emptyset$ if no ξ-name exists or a singleton $\{A_k(x_1, \ldots, x_m)\}$ if it does.

4.4 Closure Part of the System T_ξ

In this section we present tableaux deductive system T_ξ as presented in Degtyarev and Voronkov [3]. System T_ξ depends on input formula ξ (formula to be proved) in anti-Skolem negation normal form. In following sections we always assume that ξ is the input formula in anti-Skolem negation normal form. System T_ξ as presented in Degtyarev and Voronkov [3] consists of two parts. Equality solution part and tableaux part. In the first part input formula is searched for equalities and complementary literals. From search results axioms of equality part of T_ξ system are computed. In the second part inferred results from the first part are used to compute closed tableau for input formula.

4.4.1 Closures

By a closure we mean a pair $C \cdot \sigma$ where C is a clause and σ a substitution. These closures are provable objects of equality solution part of the system. We say that two closures $C_1 \cdot \sigma_1$ and $C_2 \cdot \sigma_2$ are *disjoint in variables* iff $Var(C_1 \cup C_1\sigma) \cap Var(C_2 \cup C_2\sigma) = \emptyset$. The application of a subtitution ρ to the closure $C \cdot \sigma$ is defined as $C \cdot \sigma\rho$ and denoted as $(C \cdot \sigma)\rho$.

A clause C is called *solution clause* iff it contains neither equalities nor disequalities. In particular closures containing a solution clause are objects that "we want to prove". Closures containing a solution clause are just these closures that play role in the second tableaux part of T_ξ system. The axioms of equality part of the system T_ξ are called *inital closures*. Recall that a set of ξ-names of subformula φ of ξ is either empty set or a singleton. They are generated by the following definition:

Initial closures of T_ξ system are generated according to one of the following rules:

1. Whenever a literal $s \neq t$ occurs in ξ and C is the set of ξ-names of this occurence of $s \neq t$, then the closure $s = t, C \cdot \varepsilon$ is the corresponding initial closure.

2. Whenever a literal $s = t$ occurs in ξ and C is the set of ξ-names of this occurence of $s = t$, then the closure $s \neq t, C \cdot \varepsilon$ is the corresponding initial closure.

3. Let literals $P(s_1, \ldots, s_n)$ and $\neg P(t_1, \ldots, t_n)$ occur in ξ and C_1, C_2 are their sets of ξ-names. Let the substitution σ renames variables such that variables in $P(s_1, \ldots, s_n)\sigma$ and $P(t_1, \ldots, t_n)$ are disjoint. Then the closure $s_1\sigma \neq t_1, \ldots s_n\sigma \neq t_n, C_1\sigma, C_2 \cdot \varepsilon$ is the corresponding initial closure.

Aside from the initial closures that are the axioms of the equation part, T_ξ system contains also three inference rules: two basic superposition rules and one equality solution rule. They are presented below. In the following we assume that premises of rules are disjoint in variables. Disjointness can be achieved by renaming variables in one of the premise if necessary.

Basic (right and left) superposition

$$\frac{(s = t, C) \cdot \sigma_1 \quad (u_{s'} = v, D) \cdot \sigma_2}{(u_{s'}[t] = v, C, D) \cdot \sigma_1\sigma_2\rho} \qquad \frac{(s = t, C) \cdot \sigma_1 \quad (u_{s'} \neq v, D) \cdot \sigma_2}{(u_{s'}[t] \neq v, C, D) \cdot \sigma_1\sigma_2\rho}$$

where:

1. ρ is a most general unifier of $s\sigma_1$ and $s'\sigma_2$

2. $t\sigma_1\rho \not\succeq s\sigma_1\rho$ and $v\sigma_2\rho \not\succeq u_{s'}\sigma_2\rho$

3. s' is not a variable

4. (for left superposition only) $u_{s'} \neq v$ is the leftmost disequation in the second premise.

The **equality solution** rule has the form:

$$\frac{(s \neq t, C) \cdot \sigma}{C \cdot \sigma\rho}$$

where ρ is a most general unifier of $s\sigma$ and $t\sigma$ and $s \neq t$ is leftmost disequation in the second premise.

Note that all initial closures are of the form $C \cdot \varepsilon$ where C is a clause containing only equations, disequations and ξ-names. In particular C doesn't contain any predicate symbol from ξ, because neither superpositions nor equality solution rules add such a predicate to clause part of closure. This holds also for any closure derived from initial closures. Moreover, derived solution clauses contain only ξ-names.

4.5 Tableau part of the System T_ξ

In case of the T_ξ system the *tableau* is defined as a set of clauses $\{C_1, \ldots, C_n\}$ and is denoted as $|C_1| \ldots |C_n|$ if $n > 0$. In particular for $n = 0$ we have *empty tableau* denoted by #. Such expresions are in addition to closures second provable objects of the whole T_ξ system. The tableau part of the system consists of one axiom called *initial tableau* that is of the form $|\Box|$ where $\Box$ is an empty clause. Note that this is not the same as # because $|\Box|$ is a set containing empty clause and thus n here is 1. In addition to above axiom there are also inference rules called *tableau expansions* and a *branch closure*. Note that the number of tableu expansion rules depends on the input formula ξ. Rules are defined as follows.

Tableau expansion Let D_1, D_2 and D be the sets of ξ-names of formulas φ, ψ and $\varphi \& \psi$ respectively, provided that $\varphi \& \psi$ is a subformula of ξ. Then the following is the *tableu expansion* rule:

$$\frac{|C_1|C_2| \ldots |C_m|}{(|D_1, C_1|D_2, C_1|C_2| \ldots |C_m|)\sigma}$$

where σ is a minimal subset unifier of D wrt. C_1 (we assume that the variables of the premise are disjoint from the variables of D, D_1, D_2).

Branch closure rule is defined as follows:

$$\frac{|C_1|C_2| \ldots |C_m| \quad C \cdot \rho}{(|C_2| \ldots |C_m|)\sigma}$$

where σ is a minimal subset unifier of $C\sigma$ against C_1.

Note that in branch closure C has to be solution clause because no rule introduces equality or disequality to the tableau. Moreover every derived tableu contains only ξ-names in it's clauses (on it's branches). In every expansion rule σ has an influence on D_1 and C_2 only. Furthermore, D is either an empty set or singleton of the form $\{A_i(x_1, \ldots, x_n)\}$ where x_j's have to be variables distinct with all variables in the tableau. Thus, most general subset unifier of D wrt. C_1 can be found in form $\{x_1/t_1, \ldots, x_n/t_n\}$ for some t_j's. We can say that expansion rules are *local on a branch*. On the other hand σ computed in branch closure can affect the whole tableau. So it is the only rule that can affect the variables in the whole tableau. We say that such a branch closure is *non-local*.

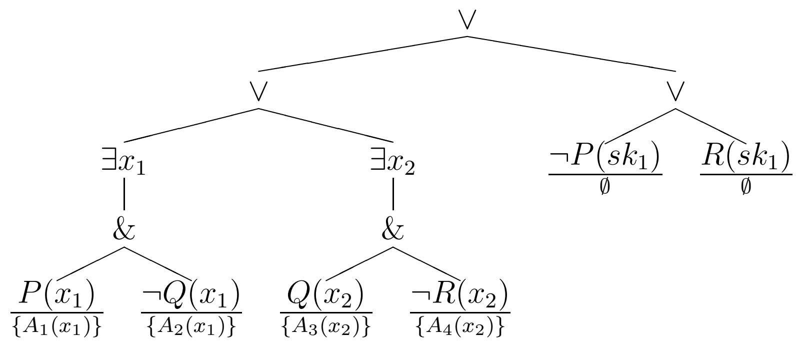
4.6 Soundness and completeness

The following theorem shows the correspondence of provability of the input formula ξ in the First-Order Logic with some derivation in T_ξ system.

Theorem (Soundness and completeness) The formula ξ is provable in the First-Order Logic iff there is a derivation of the empty tableau $\#$ in T_ξ.

4.7 Examples

We now demonstrate the proofs in the system T_ξ on two expamples. The first one is the Aristotelian Syllogism from the section 2.4. After translating the input formula into the anti-Skolem NNF we obtain the following formula ξ illustarted as a tree.



Note that we have mentioned sets of ξ-names below each literal in tree leaves. Initial closures are as follows.

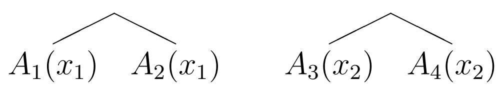
1. $\{x_1 \neq sk_1, A_1(x_1)\} \cdot \epsilon$
2. $\{x_2 \neq x_1', A_3(x_2), A_2(x_1')\} \cdot \epsilon$

3. $\{sk_1 \neq x'_2, A_4(x'_2)\} \cdot \epsilon$

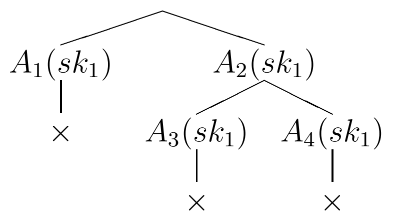
From these initial closures the following solution clauses are computed using only the equality solution rule.

1. $\{A_1(sk_1)\}$
2. $\{A_3(x_2), A_2(x_2)\}$
3. $\{A_4(sk_1)\}$

From inspecting the input formula ξ we obtain two tableau exapnsion rules. They are as follows. See also section 5.3 below and Degtyarev and Voronkov [4] for more explanation of this representation of expansion rules.



The resulting tableaux proof in the system T_ξ is in the next figure. Note that we represent the tableau proof as a tree rather than the linear representation used in the definition.

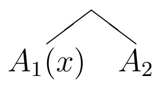


We have used in turn solution clauses 1,2 and 3 to close branches from the left to the right.

As well as the main benefit of the system T_ξ is the handling of equality we now demonstrate this system on the example taken from Degtyarev and Voronkov [4]. Let us prove the following formula in the First-Order Logic.

$$(\exists x)((g(f(x)) \neq c \vee (f(x) \neq g(x) \& (B \vee \neg B))) \vee g(g(a)) = g(f(b)))$$

There are two conjunctive subformulas with the following ξ-names. $A_1(x)$ is ξ-name of subformula $f(x) \neq g(x)$. Next A_2 is the ξ-name of the subformula $B \vee \neg B$. We have one tableau expansion rule as follows.



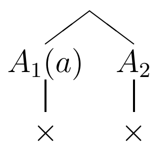
Initial closures are

1. $\{g(f(x)) = c\} \cdot \epsilon$
2. $\{f(x') = g(x'), A_1(x')\} \cdot \epsilon$
3. $\{g(f(b)) = g(g(a))\} \cdot \epsilon$
4. $\{A_2\} \cdot \epsilon$

We shall now have to use superposition rules. To do that we need reduction ordering discussed in the section 4.1.1. We shall use LPO described in the above section. To use LPO we have to fix ordering on function symbols. We consider following ordering $f > g > c > a > b$. Using the superposition and equality solution rules we have following clauses.

5. $\{g(g(a)) \neq c\} \cdot \epsilon$ [left superposition from 1,3]
6. $\{g(g(x)) = c, A_1(y)\} \cdot \{x/y\}$ [right superposition from 2,1]
7. $\{A_1(y)\} \cdot \{y/a\}$ [left superposition + equality solution from 6,5]

Note that only closures 4 and 7 contains solution clauses and after applying the substitution we will get $\{A_2\}$ and $\{A_1(a)\}$. The tableau part of the proof is very easy. We just apply these solution clauses.



Chapter 5

Implementing the T_ξ System Prover

In this section we describe our implementation of the automated T_ξ system prover. At first we present the main implementation idea. Then we present the closure solver structure. Finally we describe how the tableaux part of the system T_ξ is compiled into the Prolog program.

5.1 Basic Strategy

There are several ways to automate proof in the system T_ξ denoted in Degtyarev and Voronkov [3]. The one way is to derive closures using a saturation algorithm and then to derive tableaux proof by top-down search with backtracking as in the program leanTAP . Another way is to use a saturation algorithm for the whole system T_ξ. We have choosen the first way because from our point of view this implementation seems to be more easily extended with additional features in the future.

Our program consists of two parts. The first part is the closure solver. The main function of the closure solver is to compute initial closures and then to derive solution clauses using superposition and equality solution rules. For doing this it is necessary to implement some reduction ordering like LPO described in section 4.1.1. The second part of the prover takes solution clauses from the first part and tries to find tableaux proof for the input first-order formula.

5.2 The Closure Solver

The first part of the program is written in the object oriented programming language Python. The Python language was choosen because programming in Python is very fast and effective. Python adopts some features known from functional programming languages like lambda abstraction, `filter`, `map` and `reduce` functions or list comprehensions. Python has also built in support for lists, finite sets and associative arrays.

This part also takes care of reading the input formula from the input file. As an input file syntax the TPTP language Sutcliffe and Suttner [7] was choosen. The main reason is that this language seems to be widely dispersed and that there exists a tool `tptp2X`[1] to convert TPTP syntax files into the another theorem provers input syntax. After reading the input file the closure solver computes initial closures. Then it runs the main loop of the solver which aim is to derive solution clauses. The main loop structure is implemented in the way described in Lusk [8]. After computing solution clauses the second part of the prover is initialized. The second part of the prover is the Prolog program with a structure similar to the leanT^AP . This Prolog program structure depends on the input formula. The translation of the system T_ξ rules into that Prolog program is described below in the section 5.4.

5.2.1 Closure Solver Main Loop

Here we describe the solver main loop. The main goal of the closure solver is to infer all closures inferrable from initial closures. The main loop design is similar to the main loop in resolution based theorem provers. For more information see for example Lusk [8].

To keep generated closures in reasonable space we use the *closure subsumption relation.* If we infer new closure that is subsumed by previously generated closure, we can remove a new closure from the next computing. More information about closures subsumption can be found in Bachmair [9].

The solver works in steps. One step is only one passing the while loop below. The solver remembers two sets of closures. The set New is the set of closures inferred in the last step. The set S is the set of closures inferred before the last step. The solver starts with initial closures computed from the input formula. The solver computes in one step new closures that are inferred using superpositions or equality solution rules from closure computed in one of previous steps. To infer a closure using one of superposition rules

[1]see http://www.tptp.org/ for more information

we have to specify two closures. Then the solver computes all closures that can be inferred by one of superposition rules taking one closure from S and one closure from New as premises. Then the solver computes all closures inferred by superposition taking both premises closures from New. Note that the solver shall not try to use superposition rules for two closures from S because they are supposed to have been tried in one of previous steps. Then the solver computes all closures that can be inferred from New by the equality solution rule. Then the solver initializes the next step. At first the set S is extended by New. Finally the set New is set to hold all closures computed in the current step that are not subsumed by any closure from S. Main solver loop ends, returning the set S, when no more new closures were inffered.

```
$S := \emptyset$
$New :=$ *initial closures*
while $New \neq \emptyset$ do
    $D_1 :=$ closures inferred by superposition from $New$ and $S$
    $D_2 :=$ closures inferred by superposition from $New$ and $New$
    $D_3 :=$ closures inferred from $New$ using equality solution
    $S := S \cup New$
    $New :=$ all closures from $D_1 \cup D_2 \cup D_3$ not subsumed by one of $S$
done
return $S$
```

5.3 Compiling Expansions into the Prolog Program

In this section we are going to describe how expansions of the system T_ξ into the Prolog program are compiled. The resulting Prolog program tries to find the tableaux proof for the input first-order formula. The resulting Prolog program is generated by the closure solver after computing solution clauses. The basic idea of compiling proof search into the Prolog program is taken from Possega [10].

The resulting Prolog program consists of two parts. The first part is the same for every input formula and does not depend on results of the closure solver. The second part depends on expansions of the input formula and also contains solution clauses computed in the closure solver.

Let ξ be the first-order formula in anti-Skolem NNF to be proved. Recall that according to the definition of tableau expansion rule in the section 4.3

we have exactly one expansion rule for each subformula of ξ of the form $\varphi\&\psi$. Note that we can divide expansion rules into two subcases. The first one is that the set of ξ-names D from the tableau expansion definition is empty. The resulting expansion rule can be illustrated as follows. Note that sets of ξ-names D_1 and D_2 from the definition of tableau expansion can not be empty so they have to be singletons. We denote their only members in turn A_1 and A_2.

$$\begin{tikzcd}
& {} \arrow[dl, dash] \arrow[dr, dash] & \\
A_1 & & A_2
\end{tikzcd}$$

It can be interpreted that we can whenever branch the tableau branch putting D_1 as the left and D_2 as the right successor of the last branch node. The second case is that the set of ξ-names of is the singleton of the form $\{A(x_1, \ldots, x_n)\}$. In this case we can illustrate the expansion rule as follows.

$$\begin{tikzcd}
& A(x_1, \ldots, x_n) \arrow[dl, dash] \arrow[dr, dash] & \\
A_1 & & A_2
\end{tikzcd}$$

Note that A_1 and A_2 can also have some variables as arguments. In this case these variables are restricted to be in the set $\{x_1, \ldots, x_n\}$. This expansion can be interpreted as follows - suppose we can unify $A(x_1, \ldots, x_n)$ with some ξ-name atom allready presented on the branch using the most general unifier σ. Then we can branch this branch and put $A_1\sigma$ as the left successor and $A_2\sigma$ as the right one. We will denote the ξ-name $A(x_1, \ldots, x_n)$ as the *tableu expansion premise.* Recall that the form of σ has been discussed in the last paragraph of the section 4.3.

We shall enumerate expansion rules by natural numbers. In the Prolog program maintaining the tableau proof search we have just one Prolog clause for each expansion rule. Each such clause shall have Prolog predicate `node/5` in it's head. Everytime this `node/5` clause is entered it expands the left most tableau branch in the way described above. Whenever we expand the branch with the ξ-name A_i, and A_i is the premis of some other expansion rule, then we also insert that expansion into the list of avaible expansions. This list is hold in the fourth argument of `node/5` predicate. It has members of the form `exp(`n`,bind(...))` where `bind` is a Prolog term containing variables figuring in the expansion number n. Note that we choose some fixed order of expansions in the system T_ξ. When we have several expansions with the same expansion premise (this covers also the case of the first expansion rule type with no premise) then we apply them all at the same time.

Now we describe the arguments of `node/5` predicate. The first argument is the number of the expansion. The second is the current value of the limit controlling iterative deepening. As a completion mode discussed in the section 3.1 we use the depth of the tree. This corresponds to the number of δ-rule apllication in the leanT^AP program. The third argument is the list of ξ-names allready presented on the current (left most) branch. The fourth argument is discussed above and the fifth argument is used only in expansions with non-empty premise. In the case of the fifth argument it is the term holding current values of variables in the premise.

Aside from `node/5` clauses the resulting Prolog program contains solution clauses computed in the closure solver. Recall that the solution clause is a clause containig only ξ-names. Each computed solution clause is represented as a Prolog fact `closure/2`. The first argument is a natural number of the closure and the second argument is a list containing the Prolog representation of the solution clause.

The resulting Prolog code contains also procedures `closepath/2`, `expand/3` and `rotate/3`. First procedure tries to close the current branch using one of the closure computed in the closure solver. Procedure `expand/3` chooses the expansion from the list of allowed expansions and calls this expansion. The expansion is choosen by the procedure `rotate/3` which chooses the first member of the list and returns the list with that first member removed and added to the end of the list of avaible expansions. The resulting code also contains procedure `iterate/1` controlling the iterative deepening and `run/0` running the tableau proof search.

We illustrate this conversion on the Aristotelian Syllogism problem from the section 4.7. The corresponding `node/5` clauses are as follows.

```
node(0, Limit, Path, Exps, Bind) :-
     node(1, Limit, [a3(X_2)|Path], Exps, Bind),
     node(1, Limit, [a4(X_2)|Path], Exps, Bind).

node(1, Limit, Path, Exps, _) :-
   expand(Limit, [a1(X_1)|Path], Exps),
   expand(Limit, [a2(X_1)|Path], Exps).
```

Inferred solution clauses are encoded into the following Prolog facts.

```
closure(0, [a1(sk_1)]).
closure(1, [a3(X_9), a2(X_9)]).
closure(2, [a4(sk_1)]).
```

There is also remaining Prolog code that is the same in all Prolog programs generated by the closure solver. We call it *static code*. At first there is another static `node/5` clause before all other `node/5` clauses.

```
node(_, _, Path, _, _) :- closepath(Path).
```

This just calls `closepath` that tries to close the current path.

```
closepath(Path) :-
    closure(N, Closure),
    copy_term(Closure, FreshC),
    subset_unify(FreshC, Path),
```

There is also `expand/3` clause discussed above.

```
expand(Limit, Path, Exps) :-
    rotate(Exps, exp(N,Bind), NewExps),
    node(N, Limit, Path, Exps, Bind).
```

Finally there is the last one non-static clause running the proof search. In the case of Aristotelian Syllogism it has the following form.

```
run_limit(Limit) :-
    node(0, Limit, [],[exp(0,_),exp(1,_)], _).
```

Note that the code implementing iterative deepening is not presented here.

Chapter 6

Outline

In this paper we have studied the automated theorem proving using the tableaux methods. We have implemented the speed-up technique into the theorem prover leanT^AP . The benchmark results are presented in the Appendix C. We have also studied one expansion of the tableaux method supposed to include equality raesoning in tableaux based methods. The main benefit of this paper is the implementation of the the system T_ξ theorem prover. The benchmark results and some discussion is also presented in the Appendix C.

6.1 Future Work

We are going to implement other speed-up technique to the `xitap` prover. For example the irrelevant proof parts pruning should be simply implemented in the prover. Also some powerfull expansion selection strategy should be implemented. But the main improvement we are expecting from the computing closures in a lazy way. But nowadays it is not clear how to simply implement this improvement. Some additional future work is mentioned in the Appendix C.

Appendix A

User Documentation

A.1 Using the xitap Prover

The main benefit of this paper is the implementation of the T_ξ system prover. It can be found on attached CD-ROM in the directory `xitap`. To run the `xitap` system you need the Python and the Prolog interpreter installed on your system. Despite of the fact that the implementation can be run on every system for that Python and Prolog interpreters are avaible (in covers the most of nowaday operating systems) we recommend runnig the `xitap` system on the Linux box[1]. The implementation was tested with the Python 2.4.2[2] interpreter and with SWI-Prolog interpreter version 5.6.16[3] on Redhat Linux Fedora Core 5 operating system.

After installing Python and Prolog interpreters (that are by default installed on most Linux distributions) no additional installation is necessary. Just if you have Prolog interpreter installed in the path other than `/usr/bin/pl` you need edit the 11-th line in the file `FOL/problem.py` to fit your system (note that you can specify Windows path here). Find and edit following line if necessary.

```
PrologPath = "/usr/bin/pl"
```

To run the `xitap` system type simply `./xitap.py` on command line in the `xitap` directory. You will see following output.

[1]On other systems you shall not be able to use –challenge command line option
[2]Interpreter can be downloaded from http://www.python.org/ website
[3]Interpreter can be downloaded from http://www.swi-prolog.org/ website

```
XiTaP system - The theorem prover for the First-Order Logic
version 1.0
Usage: xitap.py [OPTION] ... [FILE]
FILE is the file in the TPTP syntax containing a problem
     to be proved.
Avaible options:
        -t, --tptp-problem=name ... prove TPTP problem,
                                    e.g. 'PUZ001+1'
        -r, --tptp-root=path    ... system path to TPTP
                                    problems library
        -n, --no-forks          ... runs as a single
                                    proccess
        -v, --verbose           ... print more information
        -r, --challenge         ... runs with 300s limit
Error: You must specify exactly one input file (or specify
       the --tptp-problem option).
```

There is also version of the TPTP library installed on the attached CD-ROM. TPTP library contains many problems to test the system. Of course you can write your own problem file. If you have the TPTP library installed in a different path you can you either use the `--tptp-root` option to specify your own path or you can edit the value of the variable `TPTPRoot` in the file `TPTPTools.py` to fit your system settings. You can try the test run of the system. Just type following (without leading `$`) to the command line from the `xitap` directory.

```
$ ./xitap.py --tptp-problem=PUZ001+1
```

Appendix B

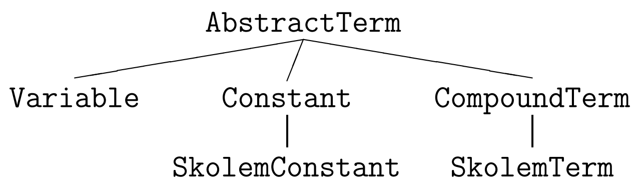
Implementation Documentation

In this section we describe the main concepts of the closure solver implementation. The structure of the Prolog program have been discussed in the section 5.3.

B.1 Main Objects Description

As well as the Python is an object oriented language the main role in the program have object classes. Here we describe the main hierarchy of classes including most important attributes and methods. Each class named `ClassName` is implemented in the separated file called `classname.py` or similar[1]. We have implemented the Python library `FOL` offering classes encapsulating basic objects of the First-Order Logic such are terms and formulas. This library can be found in the directory `xitap/FOL` on the attached CD-ROM.

We start with the description of classes implementing first-order terms. We have the following class hierarchy.



The basic abstract[2] class `AbstractTerm` contains methods supposed to be overwriten in inherited classes. We just note here that all Python methods are virtual. Most important are methods `__eq__(self, other)`, `__gt__(self, other)`,

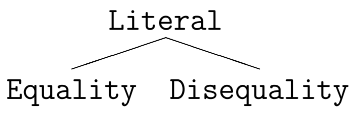
[1]For example the class `CompoundTerm` is implemented in the file `term_compound.py`

[2]Althought Python does not support abstract classes we can still use ordinary class in an abstract way. It means we do not create instances of such class.

`__ge__(self, other)` which are Python way of overloading opearators `==`, `>` and `>=`. Then when for two instances of some class inherited from `AbstractTerm` the expresion `a == b` is evaluated just the following method is called `a.__eq__(b)`. Note that for all Python class methods (except of static methods) the first argument always contains the current instance of the object[3]. We overload these opearators (in classes inherited from the `AbstractTerm`) to implement the LPO ordering from section 4.2.

Other methods implemented for all classes inherited from the `AbstractTerm` class is the method `contains`. This methods takes one argument and returns `True` if it is an instance of the class `Variable` encapsulating the first-order variable contained in the encapsulated first-order term. Next method is `unify(self, other)` which tries to unify current term (`self`) with the term specified as `other` and returns the most general unifier. We represent substitutions as associative hash arrays with the domain of variables (instances of the `Varbiable` class) and terms range. So there is no special class encapsulating substitutions. Finally every instance of each class inherited from the `AbstractTerm` contains also the method `variants(self,other)` which returns `True` if `self` and `other` encapsulates the same term up to the variable renaiming. Every term object contains also the attribute `variables` that is the set of all variables contained in the term.

Now we are going to describe classes encapulating first-order formulas. The part of class hierarchy encapulating first-order literals is in the next figure.



Note that we don't have any abstract ancestor of these classes. Except of these three classes we have also following classes encapsulating first-order formulas which are not literals: `ConjFle`, `DisjFle`, `EquivFle`, `ImplFle`, `NegFle`, `UnivQFle` and `ExQFle`. They in turn encapsulates first-order formulas which leading logical connective are conjunction, disjunction, equivalence, implication, negation and universal and existencial quantifier. Among others all these classes contains methods `nnf` and `antiskolem` which returns in turn new objects encapulating the NNF and anti-Skolem form of the original first-order formula. All of these also contains method `subformulas` providing the way of enumerating all subformulas of the encapsulated formula. They also implement the method `complement` returning new object encapsulating the

[3]This is done in an automatic way. You don't need to specify this argument when calling a method.

complement of the original first-order formula computed using de Morgan rules.

Now we shall describe classes encapsulating objects of the system T_ξ. We have a class `XiName` inherited from the class `Literal` encapsulating the ξ-name atom. In addition to all methods and attributes inherited from the `Literal` class the `XiName` contains also the attribute `xiset` containing a singleton containing self instance of the `XiName` class. In the Python code this set is created in the contructor (called `__init__` for each Python class) by the following code.

```
    self.xiset = set([self]).
```

Here the `set` is Python built-in set type constructor contructing the set instance from the list denoted in an usual way with `[` and `]` brackets. There is also the class `EmptyXiName` which is used as a ξ-name for formulas that have no ξ-name. Next class is the class `XiClause` encapsulating closures of the system T_ξ. Recall that the closure is a double containing a clause and a substitution. The clause contained in the closure contains only equations, disequations and ξ-names atoms. In our implementation we remember these in three different lists. Each list contains only corresponding literals of one type (for example just equations). Then we can identify solution clauses by testing lists containing equations and disequations to be empty lists. This is done in the method called `solvedtest`. This class also encapsulates the system T_ξ inference rules - equality solution and left and right superposition rules. The method `equality_solution` returns the new instance of `XiClause` encapsulating the closure derived by the equality solution rule if it is applic Otherwise it returns `None`. Methods `right_superpositions(self,other)` and `left_superpositions(self,other)` returns the list (possibly empty) of all closures that can be derived by using right respectively left superposition rule from closures `self` and `other`. Last but not least the class `XiClause` contains the method `variantof` implementing the closures subsumption relation.

Finally we have the class `XiFormula` implementing the translation of the input first-order fromulas into the anti-Skolem NNF and also computing ξ-names. We pass the input first-order formula to be proved by the argument of the construcotor. In the contructor we also computes expansions and initial closures of the system T_ξ. The method `solve` of the class `XiFormula` calls the main closure solver loop. It also contains the method `genprolog` which generates the resulting Prolog program implementing the tableau proof search.

The main solver loop is implemented in the source file `solver.py` and is not encapsulated in any class. It contains the function `solve(inits)` which takes initial closures computed in the contructor of the class `XiFormula` and implements the main closure solver loop as described above in the section 5.2.1.

We have also the class `Theory` encapsulating the First-Order Logic theory. We are limited to have finite number of axioms. Then the class `Theory` is just a list of axioms. The prime of the `FOL` library is the class `Problem`. This class encapsulates the first-order problem including axioms and conjectures to be proved. The main method of the class `Problem` is the method `prove` running the proof search including runnig the closure solver and the resulting Prolog program.

B.2 The TPTPTools Package

In our distribution of the `xitap` system we have also included the package allowing reading the problems wriiten in the TPTP language. It can found in t he file `TPTPTools.py` in the directory containing the file `xitap.py` not in the `FOL` subdirectory. It contains the function `ReadTPTPProblem(name)` which only argument is the TPTP problem name. The function will find the problem file according to the value of the variable `TPTPRoot` discussed in the user documentation.

It returns the instance of the `FOL.Problem` class. It also contains the function `ReadTPTPFile(filename)` which reads the input file in the TPTP syntax and returns the instance of the `FOL.Problem` class.

Appendix C

Benchmark Tests Results

In this appendix we are presenting our results of our implementation of irreflexive proof parts pruning technique and of our implementation of the sytem T_ξ. We are testing both implementations on the subset of Pelletier problems for testing automated provers taken from Pelletier [11] found in the Sutcliffe and Suttner TPTP library [7].

The implementation of the irrelevant proof parts pruning seems to be a good improvement of the original leanT^AP program. Althought we can see a little decline on some problems, that is causes by additional computation, this decline is futile in comparsion with improvent on other problems. We can say that we have same results in the worst case but better in the general case.

Also our implementation of the T_ξ system prover is tested on these problems. But we have to say that these systems are not well comparable because the main benefit of the system T_ξ is handling of equality and the leanT^AP program is known not to work well with equality. But in order to compare these implementations we have extened the problems with the First-Order Logic equality axioms. Problems containing equalities are indicated by $*$ in the problem name.

Similar approach that in our prover is used in Moser etc. E-SETHEO prover [13]. It should be interesting to test our implementation against this prover. Unfortunately we don't have working installation of the E-SETHEO prover at this moment. This and other comparsions are subjects of the future work.

In the following table the results are presented. In the first column the number of Pelletier problem is presented (see Pelletier [11]). In the second column the corresponding TPTP-name is presented. In the third, the fourth and the fifth column runtimes in miliseconds of the original leanT^AP program, leanTAP with irrelevant proof parts pruning and the `xitap` system are presented. The value ∞ means that the program did not finish after 300 seconds. More testing is the subject of the future work.

1	`SYN040-1`	55	63	166
4	`LCL181-2`	23	27	109
5	`LCL230-2`	112	118	123
10	`SYN044-1`	34	38	64
13	`SYN045-1`	42	49	230
17	`SYN047-1`	32	39	90
18	`SYN048-1`	53	57	24
20	`SYN050-1`	43	49	66
21	`SYN051-1`	63	66	157
22	`SYN052-1`	45	53	125
23	`SYN053-1`	56	59	81
24	`SYN054-1`	213	52	192
25	`SYN055-1`	43	46	176
26	`SYN056-1`	53	49	∞
27	`SYN057-1`	48	52	1645
28	`SYN058-1`	52	56	302
30	`SYN060-1`	44	34	3215
34	`SYN036-1`	2561	211	∞
38	`SYN067-1`	160	140	2023
45	`SYN069-1`	162	113	2323
47	`PUZ031-1`	∞	∞	2076
48*	`SYN071-1`	∞	∞	433
49*	`SYN072-1`	∞	∞	∞
50	`SYN073-1`	23	24	37
52*	`SYN075-1`	∞	∞	∞
53*	`SYN076-1`	∞	∞	∞

Bibliography

[1] Handbook of Automated Reasoning, A. Robinson and A. Voronkov (editors), The MIT Press, Cambridge, Massachusetts (2001).

[2] R. Hanhle, Tableaux and Related Methods, in A. Robinson and A. Voronkov (editors), *Handbook of Automated Reasoning*, The MIT Press, Cambridge, Massachusetts (2001).

[3] A. Degtyarev, A. Voronkov. Equality Elimination for the Tableau Method, *UPMAIL Technical Report 90*, Uppsala University, Computing Science Department, December 1994.

[4] A. Degtyarev, A. Voronkov. Equality Elimination for Semantic Tableaux, *UPMAIL Technical Report 90*, Uppsala University, Computing Science Department, December 1994.

[5] B. Beckert and J. Posegga. leanTAP: Lean tableau-based deduction. *Journal of the Automated Reasoning*, 15(3):339–358, 1995.

[6] B. Beckert and J. Posegga. The leanTAP-FAQ, http://emy.ira.uka.de/ posegga/leantap *(not published)*.

[7] G. Sutcliffe, C. Suttner, T. Yemenis: The TPTP Problem Library, *Proc. CADE-12*, Nancy, FRA, 1994, LNAI 814, pp. 252–266.

[8] E.L. Lusk. Controlling redundancy in large search spaces: Argonne-style theorem proving through the years. In A. Voronkov, editor, *Logic Programming and Automated Reasoning. International Conference LPAR'92.*, volume 624 of *Lecture Notes in Artificial Intelligence*, pages 96-106, St.Petersburg, Russia, July 1992.

[9] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation. *Information and Computation*, 121:172-192, 1995.

[10] J. Posegga. Compiling Proof Search in Semantic Tableaux, *Proc. Seventh International Symposium on Methodologies for Intelligent Systems*, Trondheim, Norway, June 1993 (Springer LNAI).

[11] F.J. Pelletier. Seventy-five graduated problems for testing automatic theorem provers. *Journal of Automated Reasoning*, pages 191–216, 1986.

[12] R. Nieuwenhuis, A. Rubio, Paramodulation-based theorem proving. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, The MIT Press, Cambridge, Massachusetts (2001).

[13] M. Moser, O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, K. Mayr. SETHEO and E-SETHEO. — *The CADE-13 Systems. Journal of Automated Reasoning*, 18:237-246, 1997.