Master Thesis
Software Engineering
Thesis no: MSE-2002:4
March 2002

Automated Theorem Proving

Resolution vs. Tableaux

Andreas L. E. Folkler (א)

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE-372 25 Ronneby

This page is intentionally left blank.

This thesis is submitted to the Department of Software Engineering and Computer Science at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to ten weeks of full time studies.

Contact Information:
Author:
Andreas Folkler
Address:
Myntgatan 13A
SE-417 02 Göteborg
E-mail: andreas@folkler.net

University advisor:
Dr. Bertil Ekdahl
Department of Software Engineering and Computer Science

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE-372 25 Ronneby

Internet : www.bth.se/ipd
Phone : +46 457 38 50 00
Fax : +46 457 271 25

This page is intentionally left blank.

Abstract

The purpose of this master thesis was to investigate which of the two methods, resolution and tableaux, that is the most appropriate for automated theorem proving. This was done by implementing an automated theorem prover, comparing and documenting implementation problems, and measuring proving efficiency.

In this thesis, I conclude that the resolution method might be more suitable for an automated theorem prover than tableaux, in the aspect of ease of implementation. Regarding the efficiency, the test results indicate that resolution is the better choice.

Keywords: First-order logic, tableaux, resolution, efficiency, ease of implementation

This page is intentionally left blank.

Contents

This page is intentionally left blank.

1 Introduction

Resolution and tableaux are two proof procedures of first order logic. Both methods are complete, which means that they can prove every valid argument. In automatic theorem proving, resolution is the predominant method. For example, resolution is the deductive method used in the programming language Prolog. However, the tableaux method has recently received more attention. The aim of this master's thesis was to investigate, if possible, which of the two methods is the most efficient and viable for implementing an automated theorem prover (ATP hereafter) for First Order Logic (FOL). The two methods are quite similar in that they have many inference rules in common, and what I would like to know, is why one of them (the resolution method) is predominant in automated theorem proving. Is resolution more efficient and easy to use in an ATP?

To achieve this, a program for proving theorems was to be implemented, with the ability to change method easily. During the development process, thorough notes were taken about the problems and pitfalls encountered for each proving method. The methods were then measured in aspects of performance.

1.1 The Method

Most of the literature used in this master thesis was provided by my supervisor, Dr. Bertil Ekdahl. Some of the books were found at the library or borrowed from other teachers at Blekinge Tekniska Högskola. A small part of the literature has been found on the World Wide Web. Because of this, not much of the work has been spent searching for information.

The work was to be carried out in a way similar to the iterative system development process since an application also had to be implemented. The first things to do were pre-study of the domain and the problems, and a project plan with estimated time for each activity in the work breakdown structure. The reason to implement a full application and not just write down the algorithms and calculate the performance was that many problems arise during analysis, design, and implementation. Besides that, I wanted to measure the efficiency based on real examples run through a real application.

1.2 Reading instructions

Since the literature on formal logic often uses somewhat different notation, it is recommend reading Appendix A, Glossary and Definitions, before anything else.

2 First-Order Logic

2.1 What is logic?

"Formal logic is the science of deduction. It aims to provide systematic means for telling whether or not given conclusions follow from given premises, i.e., whether arguments are valid or invalid" [JEFFREY]

Logic is the study of formalized statements, and the rules for deduction. Logic as a science was founded in 400 BC by the Greek philosopher Aristotle [KARUSH]. Nowadays, logic is very important for artificial intelligence, since this kind of intelligence often is based on deductions. In AI, logic can be used to create knowledge-based agents that reason about possible courses of action [RUSSELL].

The remainder of this chapter is divided into two parts: syntax and semantics. This is intended to be a short description of first-order logic. For more detailed descriptions and for proofs, please consult the referenced books, especially [SMULLYAN], [NERODE+], and [FITTING]. Much of the definitions and vocabulary used in first-order logic can be found in Appendix A.

2.2 Syntax

The definitions in this chapter are mostly from [FITTING], but some of them are taken from other sources. The reason for this is clarity. Not all definitions in [FITTING] demonstrate and exemplify as good. However, the consistency between definitions should have been preserved.

A language of first-order logic consists of two parts, one part that is the same for all languages (i - iv) [FITTING], and one that determines a specific language (v - vii) [FITTING]:

(i) Connectives: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$

(ii) Quantifiers: $\forall$ (for all, the universal quantifier), $\exists$ (there exists, the existential quantifier)

(iii) Punctuation: ","; ")"; "("

(iv) Variables: $x, y, z, x_0, x_1, \ldots, y_0, y_1, \ldots, z_0, z_1, \ldots$ (an infinite set)

(v) A finite or countable set R of relation symbols, or predicate symbols, each of which has a positive integer associated with it. If $P \in R$ has the integer n associated with it, we say P is an n-place relation symbol.

(vi) A finite or countable set F of function symbols, each of which has a positive integer associated with it. If $f \in F$ has the integer n associated with it, we say f is an n-place function symbol.

(vii) A finite or countable set C of constant symbols.

The first-order language determined by R, F, and C is denoted by $L(R,F,C)$, which will be written as L if there is no risk for misunderstandings.

Although this is a good start, it is not enough to know how to construct a well-formed formula. For this, we need some more definitions.

Definition (Term [HANSEN])

1. t is a variable $\Rightarrow t$ is a term.
2. t is a constant $\Rightarrow t$ is a term.
3. f is an n-ary function symbol and $t_1, \cdots, t_n$ are terms $\Rightarrow f(t_1, \ldots, t_n)$ is a term.

A ground term, or variable-free term, is a term without variables.

Next I present the definitions of atomic formula and formula, these are needed if we want to create useful sentences. E.g., a formula that is not well formed might look like $\forall \neg A(x) \wedge \exists$. It does not mean anything in our first-order language. Therefore, we need the following definitions also:

Definition (Atomic formula [NERODE+])
P is an n-ary predicate and $t_1, \ldots, t_n$ are terms $\Rightarrow P(t_1, \ldots, t_n)$ is an atomic formula.

For example, if t_1 and t_2 are terms then $\doteq t_1 t_2$ ($t_1 \doteq t_2$) is an atomic formula (of the language L). Now that we have a definition of both term and atomic formula, we can present the definition of a formula:

Definition (Formula [HANSEN])
A is an atomic formula $\Rightarrow A$ is a formula.
A and B are formulas $\Rightarrow (\neg A), (A \wedge B), (A \vee B), (A \rightarrow B), (A \leftrightarrow B)$ are formulas.
x is a variable and A is a formula $\Rightarrow (\forall x)A, (\exists x)A$ are formulas.

A sentence is a formula without free variables. To understand this, we need to have a definition of what a free variable is. A variable occurrence is called bound if it is not free.

Definition (Free-variable occurrence in a formula [FITTING])

1. The free-variable occurrences in an atomic formula are all the variable occurrences in that formula.
2. The free-variable occurrences in $\neg A$ are the free variable occurrences in A.
3. The free-variable occurrences in $(A \circ B)$ are the free-variable occurrences in A together with the free variable occurrences in B ($\circ$ is a binary connective).
4. The free-variable occurrences in $(\forall x)A$ and $(\exists x)A$ are the free-variable occurrences in A, except for occurrences of x.

When talking about quantifiers, the scope of a quantifier is often mentioned. The scope is how "far" the quantifier reaches. E.g. in the formula $(\forall x)(P(a) \rightarrow Q(x))$ the scope of x is the entire formula. However, in $(P(a) \rightarrow (\forall x)Q(x))$ the scope of x is only the subformula immediately to the right of the quantifier. "The scope of a quantifier is … the subformula in which the quantifier is main operator." ([HANSEN])

2.3 Proof systems

This section (the entire section 2.3, Proof systems) was originally written by my supervisor, Dr. Bertil Ekdahl. The reason for this is that I couldn't find the matters treated here in print.

There are several ways to prove theorems. The two investigated in this thesis represent refutation provers with no logical axioms and a set of rules of inference. Another approach to take is to establish a set of axioms, and have fewer rules of inference.

Contemporary mathematics has been strongly influenced by the axiomatic methods of Euclid. The most typical example is geometry, which was the object of Hilbert's "Grundlagen der Geometri". An axiomatic system provides a rigorous way to establish the basic principles from which mathematics can be developed. The purpose of logic is to formalize the concept of proof, i.e., defining a proof system, or equivalently, a deductive apparatus. Hilbert was the forerunner in this area and due to him, axiomatic systems are often called deductive systems of Hilbert Style. Such a system consists of three components:

(i) Logical axioms

(ii) Special axioms, or non-logical axioms, or, as I like to name them, problem-oriented axioms, because they are chosen to mirror the special field we are interested in, i.e., the special problems concerning us.

(iii) Derivation rules, i.e., the inferences that permit us to make valid arguments.

The logical axioms are those that are truly independent of the theory in question. For example, it is reasonable that $(\neg B \to \neg A) \to (A \to B)$ is independent of the actual theory. There are many ways to choose the axioms and the inference rules. Below is one example.

Axioms

1. $\varphi \to (\psi \to \varphi)$
2. $(\varphi \to (\psi \to \zeta)) \to (\varphi \to \psi) \to (\varphi \to \zeta)$
3. $(\neg B \to \neg A) \to (A \to B)$
4. $(\forall x)\varphi(x) \to \varphi(t)$ (t free for x in φ)

For a definition of free variables, see section 2.2, Syntax.

Inference rules

From φ and $\varphi \to \psi$ infer ψ (Modus Ponens).

From $v \to \varphi(x)$ infer $v \to \forall x\varphi(x)$ if x is not free in v.

There is a choice to make between what should be counted as axioms and rules respectively. Some deductive systems, as those below, have no logical axioms at all. Instead they have inference rules in which the logical axioms are "built in". Thus, the logical axioms cannot be avoided but must be considered in the rules or, as for resolution, in the formatting rules, that is, the transformation rules.

When a formal system is introduced, the question at once arises whether the initial postulated system of axioms and inference rules is complete, i.e., if it suffices to derive all the *valid* sentences. Hence, when we make our choice between logical axioms and inference rules we must always be sure that the system is complete in the sense that all valid sentences are provable.

Next, I will give three examples of deductive formal systems without logical axioms but with an enlarged set of inference rules. If we do not have logical axioms, we should be able to prove all tautologies. As an example, I have chosen the sentence $A \to (B \to A)$, which is the first axiom in the Hilbert style system above. The first example uses Natural deduction.

Natural deduction

1.	A	(assumption)
2.	B	(assumption)
3.	A	(repetition of 1)
4.	$B \rightarrow A$	($\rightarrow$ introduction)
5.	$A \rightarrow (B \rightarrow A)$	($\rightarrow$ introduction)

In (5), we no longer proceed on the assumptions (1) and (2); they are dropped.
From no assumptions, we can derive $A \rightarrow (B \rightarrow A)$. It means that the formula is true in every model.

For my second example, I use the tableaux method. The tableaux method is a refutation method (for more on refutation and contradiction, see section 2.6, Refutation).

The tableaux method

1.	$\neg(A \rightarrow (B \rightarrow A))$	($\neg$ conclusion)
2.	A	(from 1)
3.	$\neg(B \rightarrow A)$	(from 1)
4.	B	(from 3)
5.	$\neg A$	(from 3)
	$\times$	

(1) is the premise constructed as the negation of what is supposed to be a valid statement.
We arrive at a contradiction (represented by the $\times$), because we cannot conclude both A and $\neg A$. So the assumption is not true in any model, meaning that $A \rightarrow (B \rightarrow A)$ has to be true in every model. For more information on how to use the tableaux method, see section 4.2, Tableaux.

In my third and last example, I show resolution.

Resolution

Our negated conclusion, $\neg(A \rightarrow (B \rightarrow A))$, transformed to clause form:

1.	$\{A\}$	(premise)
2.	$\{B\}$	(premise)
3.	$\{\neg A\}$	(premise)
4.	$\{\ \}$	(resolve 1, 3)

Here we only use one rule of inference, the resolution rule. For more on how to use resolution, see section 4.1, Resolution. More about similarities with tableaux can be found in section 4.3, Differences and Similarities.

So, why don't we stick to formal systems of Hilbert style? The answer is that it is very difficult to see which axioms to choose and even how to combine them. In principle, it is impossible for a machine to do it. Even natural deduction has its limitation because also here we have to guess which assumption to make in order to reach the wanted derivation. The tableaux method and resolution are different since the proof methods are more mechanical. No guesses, or brilliance, are necessary.

2.4 Inference

The process of deriving conclusions from premises is called inference [GENESERETH+]. If we have a set of premises, e.g., John is either running or jumping, and John is not running, we can infer that John is jumping. The ability to make this kind of conclusion is sometimes seen as an important part of intelligence.

In more complex situations than the one in the example, it isn't always possible to draw the needed conclusion from only one inference step. These cases often require that we use several intermediate steps.

As mentioned before, to create a proof we use rules of inference. A rule of inference consists of (1) a set of sentence patterns called conditions and (2) another set of sentence patterns called conclusions [GENESERETH+]. If we have situations where a set of sentences matches the conditions, we can infer sentences matching the conclusions. An example of a rule of inference is Modus Ponens. As described in section 2.3, Proof systems, the type of system determines the number of inference rules we have. Consider e.g. resolution. It is a proving method with no logical axioms and only one rule of inference, the resolution rule (note that we might also need the factoring rule).

2.5 Semantics

A language of first-order logic is, as seen above, specified by its predicate symbols, function symbols, and constant symbols. A single language may have many possible interpretations each suited to a different context or domain of discourse [NERODE+].

For example, we have a language with just one predicate symbol, $P(x, y)$, it can then be viewed as talking about any of the following situations:

1) The natural numbers, $\boldsymbol{N}$, with P as <
2) The rational numbers, $\boldsymbol{Q}$, with P as $\leq$
3) The integers, $\boldsymbol{Z}$, with P as >

or any of a host of other possibilities. In order to specify what we are talking about, and e.g., to know how to interpret our predicate symbols, we need a model.

Definition (Model [FITTING])
A model for a first-order language $L(R, F, C)$ is a pair $M = \langle D, I \rangle$ where:
D is a nonempty set, called the domain of M.
I is a mapping, called an interpretation, that associates:
To every constant symbol $c \in C$, some member $c^I \in D$.
To every n-place function symbol $f \in F$, some n-ary function
$f^I : D^n \rightarrow D$.
To every n-place relation symbol $P \in R$, some n-ary relation $P^I \subseteq D^n$.

Apart from this model, we also need an assignment for the free variables. Otherwise, we do not know how to interpret these.

Definition (Assignment [FITTING])
An assignment in a model $M = \langle D, I \rangle$ is a mapping A from the set of variables to the set D. We denote the image of the variable v under an assignment A by v^A.

Using the model and the assignment, we can calculate values for arbitrary terms.

Example (from [FITTING])

The language L has a two-place function symbol $+$, a one-place function symbol s, and a constant symbol 0. The function symbol $+$ will be written infix. Both $s(s(0)+s(x))$ and $s(x+s(x+s(0)))$ are terms of L. Then we consider several choices of model $M = \langle D, I \rangle$ and assignment A.

1) $D = \{0,1,2,\ldots\}$, 0^I = 0, s^I is the successor function, and $+^I$ is the addition operation. Then, if A is an assignment such that x^A =3, $s(s(0)+s(x))^{I,A}$ =6 and $s(x+s(x+s(0)))^{I,A}$ =9.
2) D is the collection of all words over the alphabet $\{a,b\}$, $0^I = a$, s^I is the operation that appends a to the end of a word, and $+^I$ is concatenation. Then, if A is an assignment such that $x^A =$ aba, $(s(s(0)+s(x)))^{I,A} =$ $aaabaaa$ and $(s(x+s(x+s(0))))^{I,A} =$ $abaabaaaaa$.

2.5.1 Substitutions

A substitution is a mapping of terms for free variables. Indirectly, by unification, substitutions play an important role in automated theorem proving [FITTING]. A substitution σ that makes two formulas equal, $A\sigma = B\sigma$, is called a unifier.

Definition (Substitution [FITTING])
A substitution is a mapping $\sigma : \mathrm{V} \to \mathrm{T}$ from the set of variables V to the set of terms T.

I would like to point out that all variables are distinct and each term, t_i, in the set T is other than its mapped variable, v_i in V. Substitutions are extended to other terms in the following way:

Definition (Substitution continued [FITTING])
Let σ be a substitution. Then we set:
1. $c\sigma = c$ for a constant symbol c. (Note that $c\sigma$ also can be written as $\sigma(c)$)
2. $[f(t_1,\ldots,t_n)]\sigma = f(t_1\sigma,\ldots,t_n\sigma)$ for an n-place function symbol f.

Example: Suppose $x\sigma = f(z)$ and $y\sigma = h(a)$. Then $j(k(x), y)\sigma = j(k(f(z)), h(a))$.

When applying a substitution to a term, a new term is always produced. If two substitutions agree on the variables of a term t, they will produce the same result when applied to t. For more on the use of substitutions see section 3.1, Unification.

2.6 Refutation

Refutation is a way of proving theorems. The sentence that is supposed to be proved (the conclusion) is negated and the resulting sentence is added to the list of premises. If the resulting list is inconsistent, i.e., we reach a contradiction during our proof; then the argument[1] was valid. Both tableaux and resolution are refutation methods. A more formal description is as follows:

> **Definition (Refutation)**
> Given a set of axioms, $A_1,\ldots,A_n$, and a sentence s, the refutation theorem says that s is a logical consequence of $A_1,\ldots,A_n$, iff (if and only if) the sentence $(A_1 \wedge \ldots \wedge A_n \wedge \neg s)$ is a contradiction.

This is, as mentioned above, the basis for the technique of proof by contradiction. Refutation ensures that, if we assume the negation of a sentence and derive a contradiction, there is a direct proof (that s follows from $A_1,\ldots,A_n$) of the theorem [GENESERETH+]. This follows from the completeness theorem, see section 4.5, Soundness and Completeness.

2.6.1 Contradiction

A theorem is a provable sentence. If the negation of a sentence is contradictory, it is false in all interpretations (see section 2.5, Semantics). Then the sentence must be true in all interpretations, i.e. a tautology.

If we, during our (manually performed) proof procedure, suspect that we will not reach a contradiction, we may instead try to construct a model in which our negated theorem is valid. This means that we have produced a counter-example and that our theorem is not a tautology.

2.6.2 Logical consequence

Most of the time, we need to know more than which of our sentences are valid. We also need to know which sentences follow from which sentences, i.e., which are logical consequences (also known as logical implication).

> **Definition (Logical consequence [FITTING])**
> A sentence *X* is a logical consequence of a set *S* of sentences, provided *X* is true in every model in which all the members of *S* are true. If *X* is a logical consequence of *S*, we symbolize this by $S \models X$.

Note that I only define logical consequence for sentences. However, in the scope of this thesis, I do not need to extend it to cover arbitrary formulas. One of the most interesting features of consequence in first-order logic is that, to establish that a particular sentence *X* is a consequence of a set *S*, we only need a finite amount of the information in *S* [FITTING].

[1] Argument = the initial list of premises and conclusion, $T \models s$.

3 Further First-Order Features

This chapter discusses some features that might be necessary, or helpful, when running our proof procedures, either resolution or tableaux. Especially useful is Skolemization, together with unification, since it helps removing the existential quantifiers, and it is applicable in both resolution and tableaux. When utilizing this in the tableaux method, one inference rule can be removed right away, thus simplifying implementation. Clause form is beneficial for resolution, but it isn't needed. Since Skolemization is easier to perform when the formulas are on prenex form, it is also presented in this section.

3.1 Unification

Instead of having a time-consuming problem trying to find the correct instantiations for the universally quantified variables, unification can be used. Unification is a way of finding substitutions for variables in order to make two expressions identical. These substitutions are called unifiers.

Definition (Unifier [FITTING])
Let t_1 and t_2 be two terms (this definition extends in the obvious way to more than two terms). A substitution σ is a unifier, for t_1 and t_2 provided $t_1\sigma = t_2\sigma$. t_1 and t_2 are unifiable if they have a unifier. A substitution σ is a most general unifier if it is a unifier and is more general than any other unifer.

Here "more general" is to be interpreted in a weak sense. Hence, a unifier is more general than itself, since $\sigma = \sigma\varepsilon$, where ε is an empty substitution.

Definition (More general [FITTING])
Let σ_1 and σ_2 be substitutions. We say σ_2 is more general than σ_1 if, for some substitution τ, $\sigma_1 = \sigma_2\tau$.

Example (from [HANSEN])
The atomic formulas $P(x,a)$ and $P(f(y),z)$ are unifiable. $\sigma = \{x = f(b), y = b, z = a\}$ is a unifier, but not an mgu (most general unifier), and produce $P(f(b),a)$. $\theta = \{x = f(y), z = a\}$ is an mgu and produce $P(f(y),a)$. Here we see that $P(f(b),a)$ is an instance of $P(f(y),a)$, but not the opposite.

Suppose we have two terms, *t* and *u*, each containing variables. How do we find the set of unifiers for *t* and *u*? This is where we need to use a unification algorithm to help us determine the set of solutions for the equation *t* = *u*.

3.1.1 Unification algorithm

There are several algorithms for unification. The one presented here is the algorithm on which my implementation is based [HANSEN] (p. 317). My version is very similar but adjusted to fit the chosen language (Java) better (see also Appendix C, section 4, Unification algorithm). The algorithm tries to find a most general unifier (mgu). There can be several mgu:s but the only difference between them are the names of the variables. The concept of unification is fundamental for automated theorem proving [FITTING]. The algorithm presented is probably not one of the most efficient but is presented here because of its understandability. This was also a reason for using it in the application. However, this will not affect the comparison of the two proving methods since they use the same unification implementation.

```
Recursive Procedure Mgu(u, v)
   Begin
      u=v ==> Return({ }),
      Variable(u) ==> Return(Mguvar(u,v)),
      Variable(v) ==> Return(Mguvar(v,u)),
      Constant(u) or Constant(v) ==> Return(False),
      Not (Length(u) = Length(v)) ==> Return(False),
      Not (Part(u,0) = Part(v,0)) ==> Return(False),
      Begin i := 1,
         σ := { },
      Tag i > Length(u) ==> Return(σ),
         θ := Mgu(Part(u,i), Part(v,i)),
         θ = False ==> Return(False),
         σ := Compose(σ, θ),
         u := Substitute(u, σ),
         v := Substitute(v, σ),
         i := i+1,
         Goto Tag
      End
   End

Procedure Mguvar(u,v)
   Begin   Not Term(v) ==> Return(False),
      Occur(u,v) ==> Return(False),
      Return({u=v})
   End
```

Unification algorithm

This algorithm will unify two terms or two atomic formulas. If the two expressions are unifiable, the algorithm will return an mgu, otherwise it will return False.

Variable(u) returns true iff u is a variable.

Constant(u) returns true iff u is a constant.

Term(u) returns true iff u is a term.

Occur(u,v) returns true iff u is a variable that occurs in the term v.

Length($P(t_1,\dots,t_n)$) returns the arity of P = n.

Length($f(t_1,\dots,t_n)$) returns the arity of f = n.

Part($P(t_1,\dots,t_n)$, 0) = P

Part($P(t_1,\dots,t_n)$, i) = t_i where $1 \le i \le n$

Part($f(t_1,\dots,t_n)$, 0) = f

Part($f(t_1,\dots,t_n)$, i) = t_i where $1 \le i \le n$

Compose(σ , θ) = the composition $\sigma\theta$ of the two substitutions σ and θ.

Substitute(u, σ) = the result $u\sigma$ of applying the substitution σ on the expression u.

Example (from [HANSEN])

Unify the two terms $f(x, g(y, z))$ and $f(x, x)$. The two terms have the same arity, two, and the same function symbol f. First we unify the arguments x and x. Since they are equal the algorithm returns Mgu(x , x) = ε = $\{\ \}$. The next arguments are $g(y, z)$ and x. Since x is a variable, we get Mgu($g(y, z)$, x) = Mguvar(x , $g(y, z)$) = $\{x = g(y, z)\}$. The unifier for the two initial terms is then the composition $\sigma = \varepsilon\{x = g(y, z)\} = \{x = g(y, z)\}$. The result of the unification is $f(x, g(y, z))\sigma = f(g(y, z), g(y, z))$ and $f(x, x)\sigma = f(g(y, z), g(y, z))$.

3.2 Prenex Form

A sentence is in prenex form when all quantifiers are at the beginning of the sentence, e.g. $(\forall x)(\exists y)(P(x, y) \rightarrow Q(x))$. This can be useful when converting a formula to clause form, since the variables are standardized, and the dependencies between quantifiers are more easily detected. Similarly, the conversion to Skolem form will also be made easier if the sentence is in prenex form. One of the things that is simplified by prenex form is when a variable name occurs several times in a formula, but with different quantifiers (with their according scopes). The variables can also occur both free and bound. Although it is nothing wrong with this, it is generally simpler to avoid it, which can be done by renaming variables [FITTING]. It is important to note that a formula on prenex form is equivalent to its original sentence. There is an effective procedure for transforming any well-formed formula F into a well-formed formula A in prenex normal form such that $\vdash F \leftrightarrow A$.

If we have all variables named apart, we can begin to convert the formula to prenex form by using quantifier rewrite rules. The ones presented below is only a few (from [FITTING]):

Quantifier Rewrite Rules

$\neg(\exists x)A$	$\equiv$	$(\forall x)\neg A$
$\neg(\forall x)A$	$\equiv$	$(\exists x)\neg A$
$[(\forall x)A \wedge B]$	$\equiv$	$(\forall y)[A(x/y) \wedge B]$
$[A \wedge (\forall x)B]$	$\equiv$	$(\forall y)[A \wedge B(x/y)]$
$[(\exists x)A \wedge B]$	$\equiv$	$(\exists y)[A(x/y) \wedge B]$
$[A \wedge (\exists x)B]$	$\equiv$	$(\exists y)[A \wedge B(x/y)]$
$[(\forall x)A \rightarrow B]$	$\equiv$	$(\exists y)[A(x/y) \rightarrow B]$
$[A \rightarrow (\forall x)B]$	$\equiv$	$(\forall y)[A \rightarrow B(x/y)]$
$[(\exists x)A \rightarrow B]$	$\equiv$	$(\forall y)[A(x/y) \rightarrow B]$
$[A \rightarrow (\exists x)B]$	$\equiv$	$(\exists y)[A \rightarrow B(x/y)]$

In the table above, y is a variable not occurring on the left-hand side of the equivalencies. The slash ("/") denotes a substitution. E.g., in $A(x/y)$, the part (x/y) means that we shall substitute all occurrences of x with y in the formula A.

Example (from [FITTING])

If we have a sentence, $(\exists x)(\forall y)R(x,y) \rightarrow (\forall y)(\exists x)R(x,y)$, we must begin with renaming the variables apart. Then we get e.g. $(\exists x)(\forall y)R(x,y) \rightarrow (\forall z)(\exists w)R(w,z)$. By applying the equivalencies we can move the x and y quantifiers to the front and then the z and w ones. We then get the equivalent sentence $(\forall x)(\exists y)(\forall z)(\exists w)[R(x,y) \rightarrow R(w,z)]$. However, we can begin with moving the z and w quantifiers first, and produce the (also prenex equivalent) sentence $(\forall z)(\exists w)(\forall x)(\exists y)[R(x,y) \rightarrow R(w,z)]$.

3.3 Skolemization

Skolemization is when a sentence is transformed to Skolem form, i.e., replacing all existentially quantified variables with function- or constant symbols. These (arbitrary) symbols must not appear earlier in the proof (in neither the premises nor the conclusion). If the existential variable is not within the scope of any universal quantifier, the variable is replaced by a Skolem constant, otherwise it is replaced by a Skolem function. Skolemization of $(\exists x)(\forall y)P(x,y)$ yields $(\forall y)P(a,y)$, where a is a new constant symbol. On the other hand, Skolemization of $(\forall x)(\exists y)P(x,y)$ will produce $(\forall x)P(x,f(x))$, where f is a new function symbol. The reason for this is that the existentially quantified variable might depend on the universally quantified, and therefore we need some way of expressing this relationship. When having several existential quantifiers it is advisable to transform them from the outside and in, otherwise one might end up with unnecessarily nested functions. The following example presents this problem:

Example (from [FITTING])

Consider the sentence $(\forall x)(\exists y)(\exists z)R(x,y,z)$. If we Skolemize starting with the quantifier $(\exists y)$, we produce $(\forall x)(\exists z)R(x,f(x),z)$. Then we proceed with the quantifier $(\exists z)$, producing $(\forall x)R(x,f(x),g(x))$. On the other hand, we could have started with the quantifier $\exists z$, producing $(\forall x)(\exists y)R(x,y,h(x,y))$. Then, if we eliminate $(\exists y)$ next, we would end up with $(\forall x)R(x,k(x),h(x,k(x)))$. This is clearly a more complicated sentence than the first Skolemized version.

Definition (Skolem form [HANSEN])

Let a sentence $Q_1x_1 \ldots Q_nx_n\ B$ on Prenex form be given. $Q_1x_1 \ldots Q_nx_n\ B$ is on Skolem form $\Leftrightarrow$ all quantifiers in the quantifier block are universal, i.e., the sentence is on the form $(\forall x_1)\ldots(\forall x_n)B$.

The act of transforming a sentence to Skolem form is called Skolemization, and its definition is as follows:

Definition (The Skolem transformation [HANSEN])
Suppose $Q_1 x_1 \dots Q_n x_n B$ is not on Skolem form. Then it can be transformed to Skolem form by Skolemization. Take the leftmost existential quantifier in the quantifier block $(\forall x_1)\dots(\forall x_k)(\exists x_{k+1})\dots Q_n x_n \; B(x_1,\dots,x_n)$. Choose a new k-ary function symbol not present in B. Eliminate $(\exists x_{k+1})$ and substitute $x_{k+1} = f(x_1,\dots x_k)$ in B:
$(\forall x_1)\dots(\forall x_k)Q_{k+2}x_{k+2}\dots Q_n x_n \; B(x_1,\dots,x_k, f(x_1,\dots,x_k), x_{k+2},\dots,x_n)$
The variables $x_1,\dots,x_k$ are the ones whose universal quantifiers are to the left of $(\exists x_{k+1})$. If k=0, i.e., $(\exists x_{k+1})$ is the leftmost quantifier in the sentence, x_{k+1} will be replaced by a Skolem constant. If k>0, x_{k+1} is replaced by a Skolem function. Repeat this with the next leftmost existential quantifier until all existential quantifiers are eliminated.

The result will be a sentence on Skolem form. Note that this is defined only for sentences, not for open formulas (for a more general definition, allowing free variables, see [FITTING]). It is very important not to forget that the Skolem version of a formula may not be equivalent to the original formula, but equisatisfiable, i.e., if α is the Skolem version of β, α is satisfiable iff β ([NERODE+] p. 131).

3.4 Clauses

Clauses are beneficial for resolution. The reason for this is that they decrease the number of inference rules needed. When working with clauses, the only resolution expansion rules necessary are the resolution rule itself and the factoring rule. The definition of a clause:

Definition (Clause form [HANSEN])
Let $P_1,\dots,P_m,Q_1,\dots,Q_n$ be atomic formulas in a language of predicate calculus.
Then a clause is an expression on the form $P_1,\dots,P_m \leftarrow Q_1,\dots,Q_n \quad (m,n \geq 0)$

If $x_1,\dots,x_k$ are all the variables occurring in the clause above then it is just another way of writing the formula $(\forall x_1)\dots(\forall x_k)(Q_1 \wedge \dots \wedge Q_n \rightarrow P_1 \vee \dots \vee P_m)$. In this thesis, instead of writing the clauses as in the definition above, they will be written as $P_1 \vee \dots \vee P_m \vee \neg Q_1 \vee \dots \vee \neg Q_n$ or rather $[P_1,\dots,P_m,\neg Q_1,\dots,\neg Q_n]$.

Definition (Literal [GENESERETH+])
A literal is an atomic sentence or the negation of an atomic sentence. An atomic sentence is a positive literal, and the negation of an atomic sentence is a negative literal.

A Horn clause is a clause with at most one positive literal. If we restrict ourselves to using Horn clauses in our resolution strategy, we do not need more rules than resolution and unification [HANSEN] (we can skip factoring). This is one of the advantages restricting yourself to Horn clauses in Prolog. Another reason is that it is possible to use variants of linear resolution that are complete [NERODE+].

4 First-Order Proof Procedures

There are many proof procedures for first-order logic, e.g. Natural Deduction and Gentzen Sequents, and Hilbert systems. However, the ones of interest in this master thesis are, as mentioned earlier, resolution and tableaux. The versions presented in this chapter are more suited for hand calculation than implementation, but we chose to use them for implementation anyway (for a more detailed description of this, see section 5.2.2). The reason for this is that other, more efficient, versions are much more complex and therefore would have required too much effort to fit into the scope of this master thesis.

4.1 Resolution

Both resolution and tableaux are refutation methods. This means that they prove the theorems by showing that its negation is inconsistent with the premises.

These are the resolution expansion rules (α, β, χ are sentences):

	Affirmed	**Denied**
Not	(see the resolution rule)	$\frac{[\neg\neg\alpha]}{[\alpha]}$
And	$\frac{[\alpha \wedge \beta]}{\begin{array}{c}[\alpha] \\ {[\beta]}\end{array}}$	$\frac{[\neg(\alpha \wedge \beta)]}{[\neg\alpha, \neg\beta]}$
Or	$\frac{[\alpha \vee \beta]}{[\alpha, \beta]}$	$\frac{[\neg(\alpha \vee \beta)]}{\begin{array}{c}[\neg\alpha] \\ {[\neg\beta]}\end{array}}$
Only if	$\frac{[\alpha \rightarrow \beta]}{[\neg\alpha, \beta]}$	$\frac{[\neg(\alpha \rightarrow \beta)]}{\begin{array}{c}[\alpha] \\ {[\neg\beta]}\end{array}}$
If and only if	$\frac{[\alpha \leftrightarrow \beta]}{[\alpha \wedge \beta, \neg\alpha \wedge \neg\beta]}$	$\frac{[\neg(\alpha \leftrightarrow \beta)]}{[\neg\alpha \wedge \beta, \alpha \wedge \neg\beta]}$
All	$\frac{[(\forall x)\ldots x\ldots]}{[\ldots n\ldots]}$ (Any old name n in place of all x's.)	$\frac{[\neg(\forall x)\ldots x\ldots]}{[(\exists x)\neg\ldots x\ldots]}$
Some	$\frac{[(\exists x)\ldots x\ldots]}{[\ldots n\ldots]}$ (A new letter-name n in place of all x's.)	$\frac{[\neg(\exists x)\ldots x\ldots]}{[(\forall x)\neg\ldots x\ldots]}$
Function symbols	These yield names when all spaces are filled with names, e.g., in "(+)".	
Resolution	$\frac{\begin{array}{c}[\alpha, \beta] \\ {[\neg\alpha, \chi]}\end{array}}{[\beta, \chi]}$	
Factoring	$\frac{[\alpha, \alpha, \beta]}{[\alpha, \beta]}$	

In the table above, the horizontal dividers should be interpreted as:

Having a situation with the clauses above the divider, we can infer the clause below the divider (see example below).

Resolution in this form is very similar to the tableaux method. However, resolution is probably more known when working with clauses, or even Horn clauses as it is done in Prolog. The resolution rule is similar to a version of Modus Ponens called the Cut rule. Modus Ponens says that if we have α and $(\alpha \to \beta)$ we can infer β. The Cut rule is more general, it says that if we have $(\alpha \vee \beta)$ and $(\neg\alpha \vee \chi)$ we can infer $(\beta \vee \chi)$ [NERODE+].

Example:

Prove $\dfrac{\begin{matrix}(\forall x)(T(xb) \to T(ab)) \\ \neg T(ab)\end{matrix}}{\neg(\exists x)\,T(xb)}$

1.	$[(\forall x)(T(xb) \to T(ab))]$	(premise)
2.	$[\neg T(ab)]$	(premise)
3.	$[\neg\neg(\exists x)\,T(xb)]$	($\neg$ conclusion)
4.	$[(\exists x)\,T(xb)]$	(from 3)
5.	$[T(cb)]$	(from 4, EI c)
6.	$[T(cb) \to T(ab)]$	(from 1, UI c)
7.	$[\neg T(cb), T(ab)]$	(from 6)
8.	$[T(ab)]$	(resolve 5,7)
9.	$[\;]$	(resolve 2,8)

Here is how the proof is done. Line 4 is from line 3 by removing the double negation. Line 5 is from line 4 by existential instantiation, introducing a new constant symbol, c, to the language. Then we use line 1 to produce line 6 instantiating universally with the recently added constant symbol c. Expanding line 6 with the implication expansion rule produces line 7. Now we can use the resolution rule, line 8 is produced by resolving line 5 and 7. Then line 9, the empty clause, is from resolving line 2 and 8. Now we have produced an empty clause, then the initial list is inconsistent, and the argument was valid.

4.2 Tableaux

Similar to resolution, the tableaux method is a refutation method. A tableaux proof is a labeled binary tree, where the tree itself represents the disjunction of its branches, and each branch represents the conjunction of the formulas appearing on it [FITTING]. By setting up the tree with the premises and the negated conclusion, the method tries to close, i.e. reach a contradiction[2], all branches with several inference rules (a.k.a. tableaux expansion rules in e.g. [FITTING]) [JEFFREY]:

	Affirmed	**Denied**
Not	$\dfrac{\begin{matrix}\neg\alpha \\ \alpha\end{matrix}}{\times}$	$\dfrac{\neg\neg\alpha}{\alpha}$

[2] In the tableaux method, the $\times$ denotes a contradiction.

	Affirmed	**Denied**
And	$\frac{\alpha \wedge \beta}{\alpha}$ β	$\frac{\neg(\alpha \wedge \beta)}{\neg\alpha \quad \neg\beta}$
Or	$\frac{\alpha \vee \beta}{\alpha \quad \beta}$	$\frac{\neg(\alpha \vee \beta)}{\neg\alpha}$ $\neg\beta$
Only if	$\frac{\alpha \rightarrow \beta}{\neg\alpha \quad \beta}$	$\frac{\neg(\alpha \rightarrow \beta)}{\alpha}$ $\neg\beta$
If and only if	$\frac{\alpha \leftrightarrow \beta}{\alpha \quad \neg\alpha}$ $\beta \quad \neg\beta$	$\frac{\neg(\alpha \leftrightarrow \beta)}{\alpha \quad \neg\alpha}$ $\neg\beta \quad \beta$
All	$\frac{(\forall x)\ldots x\ldots}{\ldots n\ldots}$ (Any old name n in place of all x's.)	$\frac{\neg(\forall x)\ldots x\ldots}{(\exists x)\neg\ldots x\ldots}$
Some	$\frac{(\exists x)\ldots x\ldots}{\ldots n\ldots}$ (A new letter-name n in place of all x's.)	$\frac{\neg(\exists x)\ldots x\ldots}{(\forall x)\neg\ldots x\ldots}$
Function symbols	These yield names when all spaces are filled with names, e.g., in "(+)".	

A branch is closed when a contradiction has been found, when all branches are closed the tree is closed. As in the resolution expansion rule table, the horizontal dividers marks an inference of the formulas above the divider. E.g., the first rule in our table above means that from $\neg\alpha$ and α we can directly infer $\times$, i.e. we can close the branch. Similarly, if we have $\neg\neg\alpha$ we can infer α (in the sense that we can subjoin α to any branch passing through $\neg\neg\alpha$) [SMULLYAN].

The tableaux for first-order logic is not deterministic, i.e. the method supplies the rules we may use, but not that we must use them, or in what order. That is why there is a need for heuristics, e.g. take care of the conjunctions first etc. The following definition for tableaux is from [FITTING]:

Definition (tableaux [FITTING])
Let $\{A_1, \ldots, A_n\}$ be a finite set of formulas.
The following one-branch tree is a tableau for $\{A_1, \ldots, A_n\}$:

$$\begin{array}{l} A_1 \\ A_2 \\ \vdots \\ A_n \end{array}$$

If **T** is a tableau for $\{A_1, \ldots, A_n\}$ and **T*** results from **T** by the application of a Tableau Expansion Rule, then **T*** is a tableau for $\{A_1, \ldots, A_n\}$.

Here is an example of how the tableaux method works (same example as in section 4.1):

Prove $$\begin{array}{c} (\forall x)(T(xb) \to T(ab)) \\ \neg T(ab) \\ \hline \neg(\exists x)\,T(xb) \end{array}$$

1.	$(\forall x)(T(xb) \to T(ab))$	(premise)
2.	$\neg T(ab)$	(premise)
3.	$\neg\neg(\exists x)\,T(xb)$	($\neg$ conclusion)
4.	$(\exists x)\,T(xb)$	(from 3)
5.	$T(cb)$	(from 4, EI c)
6.	$(T(cb) \to T(ab))$	(from 1, UI c)
	$\diagup \qquad \diagdown$	
7.	$\neg T(cb) \qquad T(ab)$	(from 6)
	$\times \qquad\qquad \times$	

Here is how the proof is done. First, the tableau is set up with the premises and the negated conclusion, since it is a refutation procedure. Line 4 is produced from line 3 with the inference rule that says that if we have double negation, remove both. Existential instantiation is used in line 5 from line 4, where the x is replaced by c, which is a constant symbol new for the language. From line 1 we instantiate universally, x is replaced by c. Then we use the "affirmed only if"-rule to get line 7, which is split into two, from line 6. Since we have $\neg T(ab)$ in the same path as $T(ab)$we can close that path. On the other path, we have $\neg T(cb)$, so we can close that path also. Hence, all branches are closed and we have a closed tree. Our initial list is inconsistent, and thus our argument is valid.

4.2.1 Strictness

With strictness, we mean that a tableaux expansion rule may only be applied once to each formula, except the universal instantiations. This limitation does not affect the completeness and soundness of the tableaux method [FITTING].

4.3 Differences and Similarities

When the work on this thesis was begun, my knowledge of resolution was non-existent, but I knew how to work with tableaux. I also knew that both methods were refutation methods. At the beginning I felt that the two methods were quite different, but the more knowledge I gained, the more similar I found them to be. The only real difference is how the refutation is produced; in the tableaux method the branches are closed, while resolution produces an empty clause. My hypothesis at this point was that the two methods are very similar, and the one to choose when implementing an ATP is a free choice in terms of efficiency. This led me to believe that the things affecting the efficiency is how the heuristics are implemented. Alas, this is not true.

The tableaux implementation has a much larger obstacle to overcome in the instantiation of universally quantified variables. When we have a disjunction where the same variable is present in both the left and the right subformulas, we must instantiate them during the "expansion". The big question here is how. My first implementation did not consider this problem, and was therefore totally incorrect (see example below). The second version amended this, but it was not useful in real-world tests. The solution used at this point was instantiation by picking a random constant/function symbol from the global lists. This might be a theoretically complete solution but it was extremely inefficient. The third, and final, tableaux version, uses threads to achieve completeness. For each possible term to instantiate the above-mentioned variables, a new thread is created. The problem here is that if the theorem to prove is a little complex there will be too many threads for the computer to handle. For more on this, see section 5.2.4, Tableaux implemented.

Example of illegal instantiation

1.	$(\forall x)(P(x) \vee Q(x))$		(premise)
2.	$\neg Q(a)$		(premise)
3.	$\neg P(b)$		($\neg$ conclusion)
4.	$P(b)$	$Q(a)$	(from 1)
	$\times$	$\times$	

NOTE! Example of illegal instantiation of universally quantified variables.

This is an example of how my first version of tableaux worked. As you can see, it is not sound. It did not consider the fact that a variable must be instantiated prior to the expansion of the disjunction in which is resides.

Considering resolution, the analogous problem is how to select the two resolvents. The first implementation was accidentally a (incomplete) version of linear resolution. When this fault was fixed, I lost quite a lot in terms of performance. This second version was a breadth-first algorithm, testing all possible combinations of resolvents until an empty clause was found. Although incomplete, the first version (linear resolution) was kept for later performance comparisons.

4.4 More efficient versions

If using the procedures described in sections 4.1 and 4.2 for implementing an ATP, the resulting provers are hopelessly inefficient proving more complicated theorems. The reason for this is that it is very hard to implement a good algorithm for replacing universally quantified variables.

A way to come around this is to use unification. By making as much work as possible prior to the actual proving, the provers can be made more efficient. For resolution, the transformation to clause form is perfect for this. All the resolution expansion rules are performed during the transformation to clause form, and the only rules that are needed are factoring and resolution (if the implementation is restricted to Horn clauses only the resolution rule is needed).

Considering tableaux, not all expansion rules can be eliminated from the actual proving phase. The reason for this is the way the actual proof is done (see section 4.2, Tableaux).

By using Skolemization for removing the existential variables and then dropping the universal quantifiers, we end up with formulas perfect for unification. This is where I ran into the aforementioned problem, with disjunctions in combination with universally quantified variables. In such situations, we are faced with a problem of solving a unification of several atomic formulas at once. This is discussed further in section 5.2, Ease of Implementation.

4.5 Soundness and Completeness

The Completeness theorem ([HANSEN]):

$A_1, \ldots, A_n \vdash B \Leftrightarrow A_1, \ldots, A_n \models B$

All correct logical deductions can be made in both resolution and tableaux, i.e., they are complete. However, there are versions of resolution that are neither complete nor sound. Their lack of soundness often depends on the removed occurs check (a check done during unification). In most Prolog implementations, the completeness problem is ignored for performance reasons; the breadth-first resolution is too slow to be applicable. However, when using only Horn clauses, the incompleteness problem can be avoided, even if variants of linear resolution are used.

It is not possible to do incorrect deductions in neither resolution nor tableaux, i.e., they are sound. The following description of completeness is from [EKDAHL]:

> Gödel showed that the formal system used is complete in the sense that it captures all the valid sentences: all valid sentences are provable in the formal system used and no other formal system could do better. Gödel's formulation of completeness was "F is either $\aleph_0$-satisfiable or refutable" where F is a formula. In modern form the theorem can be stated as follows:
>
> For any set T of sentences and any formula φ, $T \models \varphi$ iff $T \vdash \varphi$, where $\models$ means logical entailment (semantic consequence) and $\vdash$ is the provability relation.
>
> Normally *any* set T of sentences is not interesting but only those which have a model. However, the theorems in a theory do not say everything about a model. For example, let N be the normal model of arithmetic and PA as before the axiom system of Peano arithmetic. Now, if $PA \vdash \varphi$ then $\mathrm{N} \models \varphi$ by the completeness theorem but if $\mathrm{N} \models \psi$ the theorem is of no help. From the view of the completeness theorem there may be sentences true in N but unprovable in PA.

It is very important to keep in mind the two different usage of completeness. One is about the axioms (as described above), the other about theories. For example, let T be a theory in a given language L. Then T is theory complete if, for every sentence φ in L, we have either $T \vdash \varphi$ or $T \vdash \neg\varphi$.

4.6 The History of Prolog

That the creators of Prolog, Alain Colmerauer and Philippe Roussel, chose resolution as their proving method was no coincidence. Prolog started out, not as a programming language, but rather as a system for processing natural language. The demands that were put on the proving method were, among others, the completeness, the risk of combinatorial explosion, and the problems for implementation on the machine. They sacrificed completeness when they chose linear resolution with unification only between the "heads of clauses". They had discovered, without being aware of it, the strategy that is complete when only Horn clauses are used. This was later demonstrated by Robert Kowalski, the father of SL-resolution together with D. Kuehner. [COLMERAUER+]

5 Evaluation

5.1 Efficiency

The first tests conducted, using a very early version of the application, did not show any differences in efficiency, but the examples used for testing were excessively small to get decisive results. Later on this changed, especially after the two complete versions were finished. The theorems (arguments) used in the tests, can be seen in Appendix B, Tested theorems.

Comparing the two complete versions (resolution and tableaux 2), we see that resolution seems to be a bit more effective. This might change if we, as described in section 7, Future Work, create a better tableaux implementation. Although, it is not likely that tableaux will be as efficient as resolution, since we, to be fair, would have to optimize that proving method as well.

5.1.1 Test results

The table below presents the results of the tests conducted. The values presented are in seconds. Each value has been calculated using the average value from ten consecutive test runs. This is done due to inexact measuring methods; it is hard to get good values on used processing time using Java. "Tableaux 2" is the complete version and "Tableaux 1" is a simpler variant. "Fail" means that the prover was unable to present any result within sixty seconds for at least three consecutive test runs. Sometimes the measured time returned from the application was zero, in those cases it was replaced by "0.01" in the calculation of average time.

	Linear Resolution	**Resolution**	**Tableaux 1**	**Tableaux 2**
Argument 1	Fail	0,276	13,582	1,647
Argument 2	0,036	0,038	0,047	0,209
Argument 3	0,04	0,041	0,105	0,2
Argument 4	0,029	0,046	0,041	0,079
Argument 5	Fail	2,019	0,242	Fail

5.1.2 Comments on the test results

As you can see in the table above, the only implementations that succeeded proving all tested theorems were Resolution (breadth-first, complete) and Tableaux 1 (simple variant, no completeness expected). Although it is no surprise that Resolution could handle all the tested theorems without problem, I was a little astonished that the complete version of tableaux failed while the simpler variant succeeded. This is probably due to a combinatorial explosion, forcing the (complete) tableaux prover to create more threads than it can handle. Therefore, we can definitely say that a better way of achieving completeness using the tableaux method is desired.

5.1.3 Test setup

Operating system: Windows 98
Java version: 2 (1.3.0)
CPU: AMD Athlon 650 MHz
RAM: 256 MB

5.2 Ease of Implementation

5.2.1 Analysis and Design

Since my knowledge of Java surpasses the knowledge I have in any other programming language, it was not a hard choice selecting language. This led me into the object-oriented technique. This was no problem, since I have experience with it and it is the one that I prefer. I do not feel this has any negative impact upon the conclusions in this thesis.

One of the first problems encountered was how to parse the arguments entered by the user. Clear instructions about how to enter a variable, function symbol, constant symbol, etc., had to be made. Another problem during analysis was how to represent the theorem as objects (in the object-oriented programming language). Since the two methods prefer the formulas to be on different formats (tree nodes and clauses) when proving, I chose to use an intermediate format for the argument. From this intermediate format, the application then transforms the argument to a format more suitable for the proving methods, i.e. clause form for resolution and tree nodes for tableaux. As for the object structure, the main concern in the beginning was whether to use inheritance or aggregate for the different parts in a formula, e.g. connectives and quantifiers. Finally, the choice was inheritance, which later proved being a good choice.

From the beginning, taking care of equality was intended. However, equality seemed to complicate things more than anticipated. As [FITTING] says: "With equality rules present, complete theorem provers have so many paths to explore in attempting to produce proofs that they can generally only succeed with simple, not very interesting theorems. It is here that good heuristics, and even machine/human interaction, becomes critical." Since the goal of this thesis was to investigate which of the two methods is the most appropriate, I thought that some sort of conclusion could be reached although equality is not included. I'm not sure of this having reached a finished application, but including equality would have made the work much harder. Despite the fact that "Virtually every mathematical theory of interest uses the equality relation…" [FITTING], there is no reason to expect that the choice to leave out equality will effect the conclusions drawn in this thesis.

All connectives, formulas, quantifiers, terms, etc., are represented as classes (for more details, see Appendix D, Design). By doing this I have several solutions where polymorphism and dynamic binding are used. E.g. when having a formula, we do not have to care about what kind it is (conjunction, disjunction etc.) to perform an action such as moving the negations as far "in" as possible. These constructs (inheritance) were also used on the proving methods, providing an easy path to expanding the prover with more methods, e.g. other versions of resolution.

5.2.2 Implementation

When I was about to start implementing the parts specific for the two methods, a small problem arose: shall the implementation use Skolemization and Unification, and thereby be more efficient than one without, or not? If the implementation of tableaux does not use unification, but the resolution version does, the comparison wouldn't be fair in either of the fields of interest (efficiency and ease of implementation). The decision was to implement both methods using Unification and Skolemization. The other alternative was to implement the whole procedure of universal and existential instantiation and the problems associated with that, such as which term to use when making a universal instantiation. As it turned out, only part of the universal instantiation could be replaced using unification. When working with tableaux, and having disjunctions with universally quantified variables, it is incorrect to postpone the instantiation of these variables (using e.g. unification later during the proof procedure). See section 4.3, Differences and Similarities, for an example. Therefore, I needed some way of making a good selection for the universally quantified variables.

At a point during the work, I had two provers that weren't quite useful, but one more efficient than the other. With about the same amount of work I think that it is safe to say that it is easier to create a somewhat useful prover using resolution rather than one using the tableaux method. The reason for this is that the lack of efficient selection mechanisms (selecting resolvents in resolution and making universal instantiations in tableaux) has more impact in the tableaux prover. However, comparing the next level of implementations is another matter. Unfortunately, the limits of this master's thesis forced me to end my efforts at this point. For ideas on what could have been done to improve the tableaux implementation, please see section 7, Future Work.

In terms of which of the two methods that were the easiest to implement, it felt like that they had about the same complexity. This feeling might be biased due to my prior experience with the tableaux method. However, if the ATP was to be fed with arguments already on clause form, resolution would have been the natural choice. On the other hand, having the arguments on normal form, many conversions could have been avoided if just implementing for tableaux. However, in this case one must implement more rules of inference, not necessarily leading to tableaux being the best choice anyhow.

In order to simplify the implementation and save some time, I chose to not implement a conversion to prenex form. Instead, the formulas are required to be on prenex form when fed to the prover. The algorithms (in pseudo code) can be found in Appendix C.

5.2.3 Resolution implemented

The first implementation used a variant of linear resolution (see [NERODE+] p. 153 and p. 66), but this was later changed to a breadth-first algorithm. The reason for this was that the linear version couldn't prove the first of the tested theorems, not within a few minutes anyway (see Appendix B, Argument 1). This theorem can be used to see whether the prover gets stuck in an infinite branch. The first version most certainly did.

To remedy this problem, the next version keeps track of all tested pairs of resolvents. All clauses are tested against each other to create an empty clause, i.e. a contradiction. This is a rather inefficient way of proving theorems, but for very small theorems, it is sufficient. To have better performance I would have had to come up with a better way of selecting resolvents. This has not been done since it is outside of the scope of this master thesis.

5.2.4 Tableaux implemented

As mentioned before (in section 4.3), the first version wasn't complete. I had made a mistake when working with universally quantified variables in combination with disjunctions. You may not postpone the instantiation (using e.g. unification later during the proof procedure) in the case where we have the same variable in both the left and the right subformulas. In the next version, this was corrected by randomly selecting a constant symbol/function symbol from the global lists (of already used constant symbols/function symbols). This is not an efficient way of proving theorems. Theoretically, it is possible to prove all valid theorems using this solution, but in reality, it is only useful for extremely small theorems. In other words: it is completely worthless. Therefore, this solution was also discarded.

The final version instead examined the branches for possible terms (using unification) to use for instantiation, and spawned a new thread for each term found. This gives us a complete, but somewhat inefficient, tableaux prover. A modified version of this, selecting term for instantiation based on points, has been included for reference.

5.3 Implementation – Size and Effort

The effective implementation time for tableaux was about thirty hours, and the amount of time required for resolution was approximately fifteen hours. Since the difference in implementation time was very small (considering the entire application), it does not matter which of the two methods one chooses in this perspective. The entire application took approximately 150 hours to complete.

Lines of code includes empty lines (added for readability) and comments. Regarding resolution and tableaux, all versions are included (two versions of each proving method).

Program part	**Lines of code**	**Hours worked (app.)**
Resolution	687	15
Tableaux	1558	30
Unification	429	N/A
Entire application	*6769*	*150*

As seen in the table above, the lines of code converge rather nicely to the hours spent on each proving method.

6 Conclusion

One thing can be said, and that is that it is not very easy to implement either one of the two methods. The perceived ease of implementation felt a little less for resolution, much thanks to its more straightforward proving, i.e. when using clauses only factoring and resolution are needed. What choice to make is not easy and careful consideration must be taken to the setting and the context. For example: Will the arguments be on clause or normal form? Do we work with first-order logic? One shall not forget that there are other proving methods than resolution and tableaux. However, if I have to make a choice, based on the setting in this thesis, resolution would be it.

Regarding efficiency, it is also very hard to draw any real conclusions. From the first tests conducted, results showed that the time required was nearly the same whether using tableaux or resolution. However, this changed when the resolution algorithm had to be remade due to the completeness problem. The new algorithm, using breadth-first resolution, was significantly slower than linear resolution or the (incorrect) tableaux implementation. Alas, this was to change again, as discussed before, when the second version of tableaux was finished. Now I had one version of linear resolution that was not complete, a complete but slow version of tableaux, and a (complete) version of general resolution. From the tests performed, we get an indication that (of the two complete versions) resolution is more efficient. Feel free to investigate the performance measurements.

As I set out on this thesis I had rather high hopes to achieve clear and unambiguous results, at least regarding the efficiency aspect. However, as time has passed, I have learnt that it is not that easy. There are so many different aspects of first-order logic, and so many considerations to make. E.g., the complexity of an automated theorem prover was much higher than expected, even when equality was left out. I hope that, although the recommendations in this thesis are vague, it will help other researchers to avoid the pitfalls and problems I have come across during my work.

7 Future Work

Among other things, the following are examples of possible future work: Extend the implementation with equality, implement more efficient versions of resolution, calculate the time needed for the different algorithms, implement more efficient versions of tableaux.

If the time would have allowed it, the tableaux should have been improved using a better algorithm for selecting instantiations for universally quantified variables. One approach is to have some kind of quantifier depth (as described in [FITTING]) allowing only a pre-selected number of instantiations.

Another possible solution is a way of investigating which possible instantiation that would help us close branches at a later point in time. This should include a way to rank the candidates for an instantiation and select the one with the highest score. However, it is very likely that a solution like this will not be complete. A quickly implemented version using this method has been included for comparison.

Acknowledgements

I would like to thank the following:

My supervisor, Dr. Bertil Ekdahl, for patience, help, and good advice.
My brother, Mårten Folkler, for reviews and ideas.
My parents, for encouragement and support.

Appendix A. Glossary and Definitions

$\Leftrightarrow$	This symbol is used to denote equivalence in the meta language.
$\Rightarrow$	This symbol is used to denote implication in the meta language.
$\vdash$	Syntactic consequence, i.e., a derived sentence in the formal system.
$\models$	Semantic consequence (a.k.a. logical consequence).
argument	"Argument" is used as in [JEFFREY], denoting a list of premises (this list can be empty) and a conclusion. The argument is either valid (not to be mistaken for the validity of formulas) or invalid.
atomic formula	From [HANSEN]: t_1 and t_2 are *terms* $\Rightarrow t_1 = t_2$ is an atomic formula. P is an n-ary *predicate* and $t_1,\ldots,t_n$ are *terms* $\Rightarrow P(t_1,\ldots,t_n)$ is an atomic formula.
clause	Let $P_1,\ldots,P_m,Q_1,\ldots,Q_n$ be atomic formulas in a predicate logic language. Then a clause is a expression on the form: $P_1,\ldots,P_m \leftarrow Q_1,\ldots,Q_n \ (m,n \geq 0)$ [HANSEN]. Let $x_1,\ldots,x_k$ be all variables occurring in the atomic formulas above. Then the clause above is another way of writing the sentence: $\forall x_1 \ldots \forall x_k (Q_1 \wedge \ldots \wedge Q_n \rightarrow P_1 \vee \ldots \vee P_m)$
completeness	A theorem proving method is complete if it can prove every valid sentence (if every valid sentence is provable), see also *soundness*. The completeness theorem ([HANSEN]): $A_1,\ldots,A_n \vdash B \Leftrightarrow A_1,\ldots,A_n \models B$
conjunction	"AND"
connective	A connective is one of the following: $\neg,\wedge,\vee,\rightarrow,\leftarrow,\leftrightarrow$
constant	A constant is a name on a specific individual in the *domain*.
disjunction	"OR"
domain	The domain is the set (in your *model*) on which you interpret all *constant symbols*, *function symbols*, *predicate symbols*, etc.
equisatisfiable	If a formula A and, e.g. a formula B that is the Skolem version of A, A is satisfiable iff B is. Note that when having this relationship, the formulas are not necessarily equivalent.
existential	see *quantifier*
FOL	First-Order Logic
formula	From [HANSEN]: A is an *atomic formula* $\Rightarrow A$ is a formula. A and B are formulas $\Rightarrow \neg A, (A \wedge B), (A \vee B), (A \rightarrow B), (A \leftrightarrow B)$ are formulas. x is a *variable* and A is a formula $\Rightarrow (\forall x)A, (\exists x)A$ are formulas.
free variable	A *variable* that is not quantified (if the variable is quantified it is bound).
idempotent	Let $\circ$ be a binary operation on $A; \circ : A \times A \rightarrow A; \circ$ is idempotent $\Leftrightarrow (\forall x)\, x \circ x = x$ [HANSEN]
iff	If and only if ($\leftrightarrow$).

implication	"If A then B "
interpretation	How the *constants*, *predicates*, *variables*, and *functions* are interpreted in the current *model*.
literal	A literal is an *atomic sentence* or the negation of an *atomic sentence* [GENESERETH+].
mgu	Most general *unifier*.
model	A model consists of a *domain* and an *interpretation*.
Modus Ponens	$A, A \rightarrow B \mid\!- B$ (its full name is Modus Ponendo Ponens)
predicate symbol	In logic, relations are named by predicate symbols. E.g. to denote that Caesar was an emperor we might write: $Emperor(Caesar)$. In this case, we have the predicate symbol $Emperor$ and the constant symbol $Caesar$, and how we interpret them is defined in the *model* of the current language.
quantifier	A quantifier is either *universal* ($\forall$) or *existential* ($\exists$), and denotes how a *variable* is viewed.
refutation	Given a set of axioms, $A_1,\ldots,A_n$, and a *formula* c, the refutation theorem says that c is a *logical consequence* of $A_1,\ldots,A_n$, iff the *formula* $(A_1 \wedge \ldots \wedge A_n \wedge \neg c)$ is a contradiction. [DRAKOS]
resolution	Resolution is a proof method using *refutation*. It can be seen as a *conjunction* of *disjunctions*.
sentence	A sentence (also called closed formula) is a *formula* without *free variables*.
soundness	A proving method is sound if it proves only the true theorems, see also *completeness*.
tableaux	The tableaux method is a proof method using *refutation*. It can be seen as a *disjunction* of *conjunctions*.
term	From [NERODE+]: (1) Every *variable* is a term. (2) Every *constant* symbol is a term. (3) If f is an n-ary *function* symbol (n=1, 2, ...) and $t_1,\ldots,t_n$ are terms, then $f(t_1,\ldots t_n)$ is also a term.
theorem	A sentence is a theorem if it is provable. Note that any sentence that occurs as an element in a proof is a theorem.
unification	When having several formulas, that seem to be alike, unification is a way of finding substitutions that make them equal. If a unifier is found it is good to find a most general unifier (mgu), i.e., the unifier that retains the most information in the formulas.
universal	see *quantifier*
valid	A formula A is valid if A is true in every model for the language [FITTING].

Appendix B. Tested theorems

This section present the theorems used for testing the different proving methods described in this thesis.

1 Theorems

Argument 1 (from [HANSEN] p. 345): This theorem can't be proved by certain versions of Prolog, since it contains one or more infinite "branches". In some cases, Prolog uses a variant of linear resolution and it gets lost in one of these "branches". Thus, this theorem is a good test for any resolution implementation that aims to be complete. The first implementation used linear resolution, and when testing the prover with this theorem it hadn't reached a result within several minutes, so it probably wasn't complete. The decision then was to re-implement the resolution algorithm, using breadth-first resolution. After re-implementation, there was no problem proving this theorem.

$$\frac{\begin{array}{c} P(a,b) \\ P(c,b) \\ (\forall x)(\forall y)(\forall z)(P(x,z)\vee\neg P(x,y)\vee\neg P(y,z)) \\ (\forall x)(\forall y)(P(x,y)\vee\neg P(y,x)) \end{array}}{P(a,c)}$$

Argument 2 (from [FITTING] p. 187): This theorem can be used to test that the Factoring rule is present in a resolution based theorem prover.

$$(\forall x)(\forall y)[P(x)\vee P(y)]\rightarrow(\exists x)(\exists y)[P(x)\wedge P(y)]$$

Argument 3 (from [JEFFREY] p. 67):

$$(\forall x)[(\exists y)L(x,y)\rightarrow(\forall y)L(y,x)]$$

Argument 4: Good for testing the unification algorithm.

$$\frac{(\forall y)(\forall z)(\forall w)(\forall u)[P(f(y,g(z)),h(b))\vee P(f(h(w),g(a)),u)]}{(\forall y)(\forall z)[P(f(h(b),g(z)),y)]}$$

Argument 5 (from [NERODE+] p. 147): Included for the same reasons as theorem number one.

$$\frac{\begin{array}{c} (\forall x)(\forall y)(\forall z)[P(x,y)\wedge P(y,z)\rightarrow P(x,z)] \\ (\forall x)(\forall y)[P(x,y)\rightarrow P(y,x)] \end{array}}{(\forall x)(\forall y)(\forall z)[P(x,y)\wedge P(z,y)\rightarrow P(x,z)]}$$

Argument X (from [HANSEN] p. 345): Included for reference, no test results will be presented. As some of the theorems above, this argument can be used to show the limits of a Prolog implementation, this time because of the omitted occurs-check. The prover is supposed to show that this argument is not provable.

$$\frac{(\forall x)P(x,f(x))}{(\forall y)P(y,y)}$$

Appendix C. Algorithms

All algorithms here are presented in pseudo-code. That should make it easy to implement regardless of your language of choice. Not all versions are presented here, I have limited it to the two, supposedly complete versions, breadth-first resolution and thread-spawning tableaux.

The source code can be submitted upon request.

1 Code in common for both proving methods

The following code is common for both methods. The argument must be in Prenex form from the beginning.

```
readFile(argument)
parse(argument)
argument.negateConclusion()
argument.negationsIn()     // move negations as far "in" as possible
argument.expandIffs()
argument.negationsIn()     // move negations as far "in" as possible
argument.expandImplications()
argument.negationsIn()     // move negations as far "in" as possible
argument.toSkolemForm()    // convert the argument to Skolem form
argument.dropUniversalQuantifiers()
argument.negationsIn()     // move negations as far "in" as possible
prove(argument)            // start proving using resolution or tableaux
```

2 Tableaux algorithm

The algorithms presented here are from the complete version of tableaux.

2.1 Method Tableaux.prove(argument)

```
convertToTreeStructure(argument)
while(not finished)
   root.expandConjunctions()

   if treeClosed()
      return VALID

   root.expandDisjunctions()

   if treeClosed()
      return VALID

   if newInstance          // if we have a new tree (we made a clone)
      resetExpansions()    // reset all instantiations so we start fresh
endwhile
```

2.2 Method TreeNode.expandConjunctions()

```
if notExpanded()        // if not expanded already, strictness
   if this.formula == conjunction
      expanded = true
      append(formula.left)
      append(formula.right)

leftNode.expandConjunctions()    // call expandConjunctions
rightNode.expandConjunctions()   // recursively down the tree
```

2.3 Method TreeNode.expandDisjunctions()

```
if notExpanded()
   if this.formula == disjunction
      variables = checkVariablesInSubformulas()
      for each variable in variables
         candidate = getCandidateTerm(variable)
         spawn new thread(candidate)
      endfor
   endif

   leftNode.expandConjunctions()    // Call expandConjunctions()
   rightNode.expandConjunctions()   // recursively down the tree

   leftNode.expandDisjunctions()    // Call expandDisjunctions()
   rightNode.expandDisjunctions()   // recursively down the tree
endif
```

3 Resolution algorithm

This is a breadth-first version of resolution.

3.1 Method Resolution.prove(argument)

```
convertToClauseForm(argument)
resolvent = findFirst()
if resolvent == null       // No resolvent found
   Return INVALID
else
   if resolvent is empty return VALID
   factoringAll()          // Check all clauses with factoring
   currentLoopLimit = -1
   size = 1
   while(true)
      size = current number of clauses
      if currentLoopLimit == size
         return INVALID    // No changes since last time, invalid
      currentLoopLimit = size
      for outer = last element to 0
         clause1 = clause at positon outer in set of clauses
         for inner = 0 to outer to size
            if inner <> outer and notTested(inner, outer)
               // keep track of which clauses that has been tested
               addTestedPair(inner, outer)
               clause2 = clause at position inner in the set of clauses
               nameApart(clause1, clause2)
               resolvent = resolve(clause1, clause2)
               if resolvent <> null
                  add the (unified) resolvent to the set of clauses
                  if isEmpty(resolvent)
                     return VALID
                  endif
               endif
            endif
            if timeOut()
               return TIMEOUT
            endif
         endfor
      endfor
   endwhile
endelse
```

4 Unification algorithm

Using a variant of the unification algorithm described in [HANSEN] p. 317

4.1 Method mgu(Atomic atom1, Atomic atom2)

```
if atom1.match(atom2) == exact or atom1.match(atom2) == contradiction
   return empty substitution
else if atom1.match(atom2) <> no match  // it may be possible to unify them
   return mgu(atom1.predicateSymbol, atom2.predicateSymbol)
else
   return null
```

4.2 Method mgu(PredicateSymbol ps1, PredicateSymbol ps2)

```
if ps1.match(ps2) == exact
   return substitutionList
else ps1.match(ps2= <> no match
   nameApart(ps1.termList, ps2.termList)
   return mgu(ps1.termList, ps2.termList)
else
   return null
```

4.3 Method mgu(TermList t1, TermList t2)

```
Vector localList
for i = 0 to t1.length          // t1.length == t2.length
   result = mgu(t1.termAt(i), t2.termAt(i))
   if result == null
      return null               // the two terms could not be unified
   localList.add(result)
   substitute(t1, localList)
   substitute(t2, localList)
return localList
```

4.4 Method mgu(Term t1, Term t2)

```
Vector localList
if t1 == t2
   return empty substitution
else if t1 is a variable
   return mguvar(t1, t2)
else if t2 is a variable
   return mguvar(t2, t1)
else if t1 is a constant symbol or t2 is a constant symbol
   return null
else                 // both are function symbols
   if the function symbols have different names
      return null
   else
      tmpResult = mgu(t1.termList, t2.termList)
      if tmpResult == null
         return null
      localList.add(tmpResult)

return localList
```

4.5 Method mguvar(Variable v, Term t)

```
Vector localList
if t is a function symbol
   if t.occurs(v)
      return null
localList.add(new Substitution(v/t))
return localList
```

Appendix D. Design

The class design was refined during the implementation, since it is impossible to think of all possible solutions during the initial design and analysis. After a while, during the implementation phase, I realized that the way I had implemented negations wasn't flexible enough. However, the new solution did not affect the class diagram. Where the class diagram denotes a one-to-many relationship, it is mostly implemented using vectors (java.util.Vector). This was done because of simplicity and reuse. If I had had more time to make a redesign, there would probably have been some more subclasses to the class Formula, e.g. a class Unary, working as a super class for Quantified and Negated.

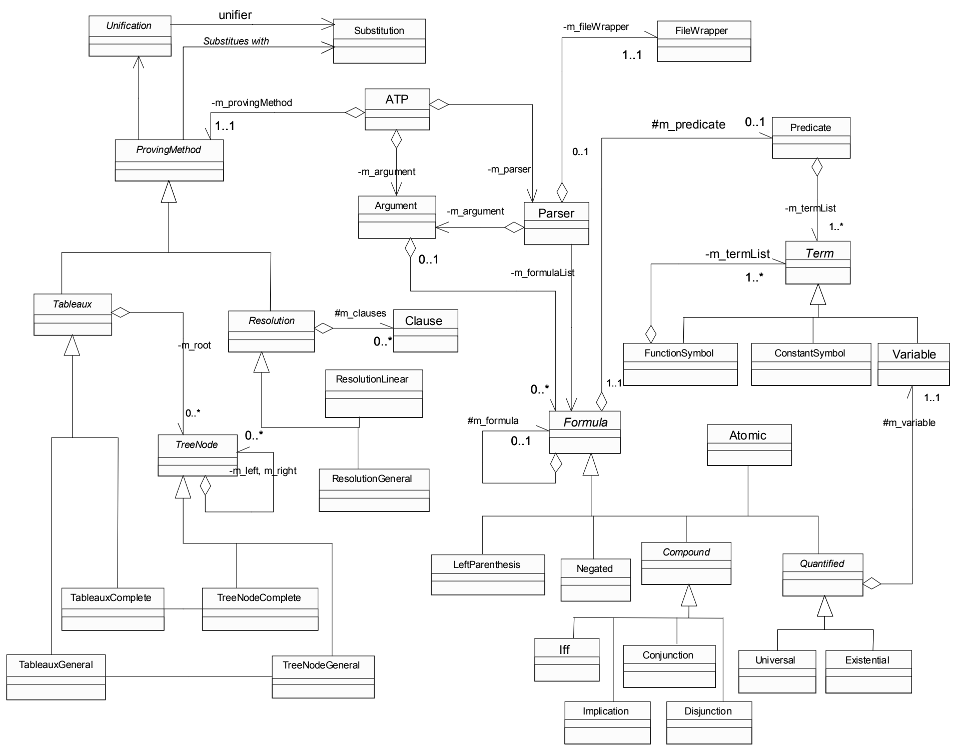

Figure 1: Class design

References

[COLMERAUER+]	Alain Colmerauer, Philippe Roussel; *"The birth of Prolog"*, *"History of programming languages"* (editors Thomas J. Bergin, Richard G. Gibson); © 1996 ACM Press; ISBN 0-201-89502-1
[DRAKOS]	Nikos Drakos; "*Logic with PAIL*"; June 25 1991; http://cbl.leeds.ac.uk/nikos/pail/logic/subsection3.3.6.html
[EKDAHL]	Bertil Ekdahl; *"Interactive computing does not supersede Church's thesis"*; 1999; ISBN 0-9668650-5-7; p. 261-265
[FITTING]	Melvin Fitting; *"First-Order Logic and Automated Theorem Proving"*; 1996, 1990 Springer-Verlag New York, Inc.; ISBN 0-387-94593-8
[GENESERETH+]	Michael R. Genesereth, Nils J. Nilsson; *"Logical Foundations of Artificial Intelligence"*; 1987 Morgan Kaufmann Publishers Inc.; ISBN 0-934613-31-1
[HANSEN]	Kaj B. Hansen; "*Grundläggande logik*"; Studentlitteratur 1994; ISBN 91-44-37002-4
[JEFFREY]	Richard Jeffrey; *"Formal Logic: Its scope and limits"*; Third edition 1991 McGraw-Hill, Inc; ISBN 0-07-032357-7
[KARUSH]	William Karush; *"Matematisk uppslagsbok"*; Wahlström & Widstrand 1970; ISBN 91-46-13004-7
[NERODE+]	Anil Nerode, Richard A. Shore; "*Logic for Applications*"; 1997 Springer-Verlag New York, Inc.; ISBN 0-387-94893-7
[RUSSELL+]	Stuart Russell, Peter Norvig; *"Artificial Intelligence – A Modern Approach"*; Prentice-Hall Inc. 1995; ISBN 0-13-360124-2
[SMULLYAN]	Raymond M. Smullyan; *"First-Order Logic"*; Springer-Verlag Berlin Heidelberg New York 1968