NON-CLAUSAL RESOLUTION

Theory and practice

Hashim Habiballa

Department of Computer Science
Faculty of Science
University of Ostrava,

Ostrava, The Czech Republic.

October 2000

CONTENTS

Chapter 1

1 Introduction.

The first-order theory is well known notion among mathematicians and computer scientists. It is a suitable formal representation for expressing knowledge and deducing theorems. Although it is very popular and clear way, the problem of automated theorem proving is not so simple. If you want to enjoy the power of logic, you have to perform many transformations, before you can use any deductive system. These transformations destroy the meaning of the formula and may produce high amount of clauses (in case of clausal resolution). Of course, it has decisive advantages such as the efficiency of the proof. Nevertheless, I believe from the theoretical point of view it is a very interesting investigation to search for a deductive system, which is free of the need of destructive transformations.

First-order logic covers many deductive systems and methds. In spite of high diversity of these systems, most of them have one essential fault. They are determined to narrow class of formulas. Clearly there is for example the method of semantic tableaux, which is not disadvantageous in this manner. This method provides us with complete process of proving without the need of transformations. It brings set of rules for decompostion of a formula into several branches, of which we can decide, if they are closed or not. This procedure enables to decide about inconsistency of a formula, but such a proof will not be of our interest. We will be looking for a classical theory with set of special and logical axioms together with inference rules. The proof is then supposed to be a sequence of formulas, which are either axioms or derived from them by usage of inference rule. And as it is obvious in automated theorem proving, we will stand on refutational proving i.e on proofs based on negated goals leading to denial of a set of axioms and goal. Then it is not surprising, that the focus of the thesis slip into a theory of resolution. The conventional view of resolution is closely related with clausal normal form of formula and with skolemization in predicate logic. It is not very difficult to obtain a clausal normal form of any formula (particularly for non-quantified formulas), however as it was written, result is logically equivalent (with respect to satisfiability), but it is unacceptable with respect to the structure of original statement.

I was disappointed from this fault and I looked for some solution to this problem. There was a lot of papers on the Internet concerning to resolution and among these papers I found a technical report “A theory of resolution” [Ba97], which presents detailed exploration of the possibilities, how to perform deduction on skolemized formulas of predicate logic. Though the paper describes far more than the base extension of resolution rule, I used the paper mainly as a source of inspiration for further research of strategies for suppressing the redundancy of an inference and handling existential variables. The controversy, if the exploration of generally valid resolution rule is valuable, may be answered by following simplified analogy. In mathematics there are exact analytical methods and numerical methods for similar problems. At first sight one can say that the first sort of methods is not significant, because numerical ones are more universal, eficient and simpler for automated utilization. Nevertheless I’m not in doubt that everybody understands benefit of further research of analytical approaches.

But the theoretical contribution is not the only argument. After the proving application became functional, the fact, that it can prove some elementary theorems almost instantly and clausal prover spent a lot of time on them, was logical, but surprising. In the section concerned to examples you will find that a relatively short sequence of equivalencies may flood a clausal prover with resolvents and the non-clausal prover solved the problem in one resolution step (with no more than 5 redundancy check steps). Clearly I cannot base the advocacy of a methodology in one or several cases, however I would like to criticise such positions that put forward the efficiency of the proof and simultaneously allow to solve simple problems inefficiently. The complexity never can be perfect but only optimal and that's why every approach has to be considered, even if it seems to be not universal.

The inseparable part of this thesis is the computer application, which has to illustrate presented resolution techniques. This application is not a full coverage of theoretically proposed deductive system. I hope that it is together with the mentioned theoretical extensions a small contribution to the knowledge in automated theorem proving. It should provide the reader with a tool for his test with general resolution as well as test of resolution strategies. I ask users of the system to be benevolent to it. Please, realise that the program was elaborated and tested by one person under the pressure of study and professional duties. It is only co-objective of the thesis and that's why it can't be compared with systems designed by a team of theoretical specialists and programmers.

The second chapter introduces some common notions (first-order logic, deductive systems). I expect the thesis is intended for mathematicians and computer scientists, who are familiar with basics of logic, deduction and resolution. Therefore these introduction are straight without useless details. Chapter three defines general resolution and brings some examples, it also contains some modifications and extensions improving standard definitions both undertaken and self-devised. In this section there are some proofs especially the proof of completeness of general resolution, which tries to be original and so it could be a hole of the thesis. Chapter four shows the problem of high amount of resolvents generated during inference process and gives some common solutions and consequence checking specially modified for the purposes of this thesis. General resolution gives a good chance to detect redundant formulas by its own power. The fifth chapter describes application data structures and algorithms used for inference process in detail. It uses the Pascal programming language to demonstrate algorithmical solutions and Pascal comments to make the source code clearer. Since the source code exceeds 100 KB and this thesis is presented at mathematical department, the chapter is as short as possible. The sixth chapter produces the description of the computer application called GEneralized Resolution Deductive System (GERDS). GERDS supports my research in resolution techniques and it is not user application in the right meaning. The sufficient and brief guide to the GERDS is given in this section. It is also controversial section however needful for practical automated theorem proving with general resolution. The section seven will show some examples of general resolution, which is probably the most suitable way to understand methods and power of general resolution. It tries to show off interesting properties of general resolution and its advantages and disadvantages in comparison with common resolution rule.

The main intended contribution of the paper is to extend and illustrate previously founded generalizations of non-clausal deduction. It should bring the general deductive system, which may process general formulas of predicate logic and the only actual need is the notion of extended polarity, which do not require any transformation, but only counting of some characteristics of every subformula. Such a counting can be performed during the construction of the parse tree, that is the most suitable representation for computer processing anyway. If we consider only non-existential formulas, then the resolution is completely trivial and does not require anything special. The problem of existency in conjuction with equivalence is an open problem and gives space for further work..

I decided to write the thesis in English, even my skills are low, since I hope that it makes the paper more understandable and therefore there is as few useless words as possible.

Chapter 2

2 Preliminaries.

2.1 First-Order Logic.

Before we start with the explanation of the general resolution, it is necessary to introduce some common notations from first-order theory. It will be used the notation close to logic programming. At first, it has to be shown the alphabet of the first-order predicate logic language. Brief and pregnant explanation can be found in [Kl67] , [Ri89] or [No99]. It consists of:

- Variables: Character string starting with a capital letter or underscore; containing alphanumeric character or underscore, e.g. My_First_Var, _trash1.
- Functors and predicate names: Character string starting with a lower case letter e.g. sqrt, is_a_Child. It is not actually needed to use a separate notion of the constant, because they can be treated as 0-ary functors.
- Logical connectives: $\land$ - conjunction, $\lor$ - disjunction, $\rightarrow$ - implication, $\leftrightarrow$ - equivalence, $\neg$ - negation.
- Logical constants: $\bot$ - false, $\top$ - true. It is not used any special symbols in the source set for logical constants in the application, but there is a flag indicating logical value of the subformula in the result and it is represented by $\bot$ and $\top$ too.
- Quantifiers: $\forall$ - universal and $\exists$ - existential.
- Special symbols: (,) , and **$\geq$, $\leq$, <, >, =, $\neq$.** The comparing characters have no special handlers and serve as predicates for user usage.

The best way to define the language of the predicate logic (PL) is the introduction of the grammar in Backus –Naur Form:

Definiton 2.1.: Syntax of predicate logic by BNF.

<Formula> ::= <Imp> { $\leftrightarrow$ <Imp> }

<Imp> ::= <Dis> { $\rightarrow$ <Dis> }

<Dis> ::= <Con> { $\lor$ <Con> }

<Con> ::= <Subformula> { $\land$ <Subformula> }

<Subformula> ::= $\neg$ <Subformula> | <Quantifier section> <Subformula> | '[' <Formula> ']' | <Predicate>

<Quantifier section> ::= <Quantifier character> <Variable> { , <Variable> } {<Quantifier character> <Variable> { , <Variable> } }

<Predicate> ::= <Predicate name> <List of parameters> | <Term> <InfixPred> <Term>

<Term> ::= <Term2> { <+/- operator> <Term2> }

<Term2> ::= <Base> { <*// operator> <Term2> }

<Base> ::= <Variable> | <Function> | <StrLit> | <Number> | + <Number> | - <Number> | (<Term>)

<Functor> ::= <Lower case> {<Alphanumeric>}

<Predicate name> ::= <Lower case> {<Alphanumeric>}

<List of parameters> ::= { (<Term> { , <Term> }) }

<Function> ::= <Functor> <List of parameters>

<Variable> ::= <Upper case> {<Alphanumeric>}

<Number> ::= <Integer> {. <Integer>} { e <+/- operator> <Integer>}

<StrLit> ::= " {<Alphanumeric>} "
<InfixPred> ::= ≥ | ≤ | < | > | = | ≠
<Quantifier character> ::= ∀ | ∃
<+/- operator> ::= + | - , <*// operator> ::= * | /
<Lower case> ::= a | .. | z, <Upper case> ::= A | .. | Z | _
<Alphanumeric> ::= <Lower case> | <Upper case> | <Numeric> , <Integer> ::= <Numeric> { <Numeric> } , <Numeric> ::= 0 | .. | 9

As you could see, the language is common. It is constructed of atoms connected by logical connectives and incorporating quantifiers. Atoms are predicates (standard or infix) and they are represented by predicate names and list of parameters e.g. child(mary, john), where child is a predicate name and mary and john are parameters. It is important to mention, that the difference between function and predicate (in logical meaning) may be a source of confusion, because they can start with lower case letter both. That's why function can refer to both predicate and term. However, there is not a possibility to mistake the term for a predicate in a formula, because when we reach the level of atom with name starting with lower case letter, it is a predicate, and all the inferior levels must contain only terms. We can also understand a predicate as a function returning logical value, if we feel something wrong in this notation of syntax. Logical constants are passed away, since they aren't needful.

It fully satisfies the definition of the syntax from the programmer's view and now we can give mathematical definitions of well formed formulas of PL.

Definition 2.2.: Terms of predicate logic.

i. A variable X or nulary functor (constant) c is an (atomic) term.
ii. Let f be a n-ary functor, +, -, *, / be 2-ary functors and t_1, ..., t_n be terms. Then the expressions $f(t_1, .., t_n)$, $t_1 + t_2$, $t_1 - t_2$, $t_1 * t_2$, t_1 / t_2 , (t_1) are terms.

Definition 2.3.: Formulas of predicate logic.

i. The logical constants $\bot$, $\top$ are (atomic) formulas.
ii. Let p be an n-ary predicate name and t_1, .., t_n be terms, then the expression $p(t_1, .., t_n)$ is an (atomic) formula.
iii. If A and B are formulas then $A \wedge B$, $A \vee B$, $A \rightarrow B$, $A \leftrightarrow B$, $\neg A$, [A] are fomulas.
iv. If X is variable and A is formula then $\forall X A$ and $\exists X A$ are formulas.

(Atomic formulas or its negations are obviously called literals.)

Now we must discuss some problems related to it. The first problem results from the usage of existential variables. The base of non-clausal resolution described on [Ba97] requires only ground cases of formulas i.e. skolemized formulas with variables substituted by a term without variables. The fully unrestricted resolution demands some restrictions to substitution of terms into variables. These notations are assumed to express occurence and substitution: E[E'] means that the expression E contains E' as a. The result of simultaneous replacement of all occurrences of E' by E'' is denoted by E[E'/E''] . They are also considered partial substitutions – E[E'|E''] represents replacing of one occurrence of E' by E''.

Because an unusual method of a substitution is given in the application, there is not a problem with free and bound variables like it is obvious in standard PL. In PL we speak about substitutibility of a term into a variable. The term t is substitutible to X if there is not a variable in t which could become bound after substitution. In the application every bounded variable is considered to be unique object with its memory address and it can't be mistaked by another variable with the same name e.g. $\forall X\ a(X) \vee \exists X\ b(X)$, where we have two different X variables. If we substitute X from b(X) into X from a(X), it is an invalid operation, because existential meaning of X became universal. But it only occurs if we print this result to the user. Internal representation of a formula handles unique memory addreses instead of names for example the first. Although it is simply solved, there still is the question of substitution existential variables into universal ones. It was found solution, which will be discussed below, but for short it can described as complete checking, if all the variables over its scope are assigned a value e.g. We have $\forall X \exists Y\ p(X,Y)$ and we can assign Y anywhere only if X has assigned a value. When you examine it, you will find, that it is the right meaning of existential variable. Existence of such Y is strictly depending on specific X.

Now we briefly summarize the semantics of first-order logic. First it is introduced the notion of Interpretation.

Definition 2.4.: Interpretation structure and rules for language of PL.

The interpretation structure is $M = \langle D, \mathbf{f_1}, .., \mathbf{f_n}, \mathbf{p_1}, .., \mathbf{p_n} \rangle$, where D is a non-empty set called universe, $\mathbf{f_k}$ represents functions, used in formulas, of the form $f : D^p \rightarrow D$ and $\mathbf{p_k}$ represents relations of predicates $p \subseteq D^p$;to each functor and predicate name is assigned appropriate object from D, we can call this mapping Denot. Further it is defined mapping e as variables evaluation from the set of all variables into an object of D.

Value of term t in M with evaluation e – t[e] is:

i. if t is a constant c, c = **c**, where c as a language expression refers to an object **c** from D.
ii. if t is a variable X, then t[e] = e(X).
iii. if the term is of the form $f(t_1,..,t_n)$, then $t[e] = \mathbf{f}(t_1[e],..,t_n[e])$, where $\mathbf{f}$ = Denot(f).

for infix operators of +, -, *, / we may consider Denot mapping to assign standard arithmetic functions.

We define, that the formula F is true in M with evaluation e - $M \models F[e]$ as follows:

i. if F is a predicate of the form $p(t_1,..,t_n)$, then $M \models F[e]$ holds iff $(t_1,..,t_n) \in \mathbf{p}$, $\mathbf{p}$ = Denot(p).
ii. if F is of the form $\neg G$ then, $M \models F[e]$ iff $M \models G[e]$ doesn't hold.
iii. if F is one of the form $G \wedge H$, $G \vee H$, $G \rightarrow H$, $G \leftrightarrow H$, then $M \models F[e]$ holds, depending on the connective:
   $G \wedge H$: iff $M \models G[e]$ holds and $M \models H[e]$ holds,
   $G \vee H$: iff $M \models G[e]$ holds or $M \models H[e]$ holds,
   $G \rightarrow H$: iff $M \models G[e]$ doesn't hold or $M \models H[e]$ holds
   $G \leftrightarrow H$: iff $M \models (G \rightarrow H)[e]$ holds and $M \models (H \rightarrow G)[e]$ holds
iv. if F is of the form $\forall X\ G$, where G is a formula of the language, then $M \models F[e]$ holds iff for every object $m \in D$: $M \models G[e]$ holds, where e(X) = m .
v. if F is of the form $\exists X\ G$, where G is a formula of the language, then $M \models F[e]$ holds iff there is a object $m \in D$: $M \models G[e]$ holds, where e(X) = m.

Formula F is *satisfiable in M*, if for some e $M \models F[e]$ holds. F is *satisfied (valid) in M* - $M \models F$, if $M \models F[e]$ for every e. If the formula is satisfied in every interpretation, then it is *(logically) true*.

Clausal form of a formula is a notion, which we will use in definitions and try to avoid, so it is reasonable to introduce it.

Definition 2.5.: PL-formula in clausal form.

Formula of predicate logic is in clausal form if it is of the form $\forall X_1..\forall X_n$ [A], where:

i. $X_1..X_n$ are all of the variables from formula A

ii. A is in the conjunctive normal form (CNF) i.e. A is conjunction of finite number disjuncts and disjunct is a disjunction of finite number literals, where there is no mutually complementar pair of literals in each disjunct.

Theorem 2.1.: Existence of clausal form for PL-formula.

For every PL-formula A there is a formula B in clausal form, where A is satisfiable iff B is satisfiable.

2.2 Deductive systems and principles.

Let's have a look to two deductive systems of first-order logic to see the advantages and disadvantages of them. A detailed description of these systems can be found in [Lu95] or [Ce81]. Deductive (Axiomatic) system consists of :

1. Language.
2. Axioms – source formulas (schemas) for inferring theorems.
3. Rules - enabling to derive theorems from axioms.

Since it is a basic subject matter, it will not be described exact grammar of a system. And by reason that we are interested in theorem proving from the set of special axioms, we stay on discussing about the construction of formulas and inference rules.

First let's stop with the Hilbert's axiomatic system.

It has two allowed connectives – negation and implication. Although it is known that negation and implication forms complete set of connectives i.e. every formula can be rewritten to it, the lucidity of the proof is low. We can dispute about some special cases such as Horn clauses: $a_1 \wedge .. \wedge a_n \rightarrow b$. They are simply and clearly transformable into $a_1 \rightarrow(.. \rightarrow (a_n \rightarrow b))$, but some simple cases with equivalence or negation in superior levels like $\neg(a \vee b)$ $(\Rightarrow \neg(\neg a \rightarrow b))$ have the meaning of the formula hardly recognizable. The first transformation is quite close to Hilbert style: " if a_1 holds and .. and a_n holds then b holds too" transforms to "if a_1 holds then if .. then if a_n hold then b holds too. The second one is recondite "it doesn't hold a or b" (in the other words a doesn't hold and b doesn't hold) transforms to " it doesn't hold that if a doesn't hold then b holds".

In the other hand the modus ponens rule looks smart, as we can understand it : "if holds the theorem of the form - if a then b and if a holds, then b must hold too." The axiom of specification ensures the possibility of transformations of formulas into their ground cases.

The second deductive system uses the best known principle, it is the resolution deductive system with the resolution principle. Language of the system accepts formulas in conjunctive normal form. The resolution rule is notoriously known:

Consider two clauses of the form $C_1 \vee l$ and $C_2 \vee l'$, where l and l' are mutually complementary. Then it can be deduced from the two above clauses the clause $C_1 \vee C_2$. This type of a rule also has a reasonable sense. Let's take the modus ponens rule and try to see it as a specialization of the resolution rule. $A \rightarrow B$ could be rewritten to $\neg A \vee B$ and that's why the MP rule has the resolution form: $\neg A \vee B$ and A faces to B. The resolution rule can be also considered in the implicative form and then it can viewed as transitive rule $A \vee B$, $\neg B \vee C \Rightarrow \neg A \rightarrow B$, $B \rightarrow C$, which gives $\neg A \rightarrow C$ that is $A \vee C$. So the complementary couple of atoms is redundant and can be omitted, if we construct new implicative theorem. It consents to the clausal meaning. The two complementary atoms in the conjunction have no gain, because there is no model depending only on these two literals. So that's why every model of $(C_1 \vee l) \wedge (C_2 \vee l')$ on C_1 or C_2 only. In the other words $(C_1 \vee l)$ and $(C_2 \vee l')$ must be both true in such model, but then C_1 must be true if l is false or C_2 must be true if l is true and no other case exists. This is a little less clear explanation than an implicative form, I'm convinced.

When we considered these two systems, we didn't speak about predicate logic modifications of these systems deeply. It was a wilful omission. These extended versions do not require a lot of effort to devise. It is the question of finding the right way to make formulas ground and to handle existence. The first one is solved with unifiers (in resolution based systems) or rules of specialization (in Hilbert system) and the second one is solved by skolemization (transformation of existential variable to a new function with superior variables as arguments) or implicitly by special functors (in Clausal Form Logic).

3 General Resolution.

3.1 Definitions and examples.

When we use refutational theorem proving, we deduce new formulas from given ones and negated goal and search for a contradiction. The widely used inference rule is resolution, originally introduced by Robinson. Now we present the results from [Ba97] to show the power of general resolution that applies to general formulas. Because hereinbefore we discussed that it is possible to stay on propositional case and then only find suitable unification method to extend it, let's start with propositional forms of rules as presented in [Ba97].

For the purposes of mentioned article, there were introduced some notions. Inference rule is an n-ary relation on expressions, where n ≥ 1. The elements of such relation are written as

$$\frac{E_1 \,..\, E_{n-1}}{E}$$

and called inferences. The expressions E_1 .. E_{n-1} are called premises, and E is the conclusion, of the inference. An inference system is a collection of inference rules.

An inference is sound if the conclusion is a logical consequence of the premises, i.e., E_1 .. E_{n-1} $\models$ E . The following definition of resolution for formulas is sound.

Definition 3.1: General resolution.

$$\frac{F[G] \quad F'[G]}{F[G / \bot] \vee F'[G / \top]}$$

It is the resolution on G and the conclusion of the inference is called resolvent of the two premises. It is also called F the positive, F' the negative premise, and G the resolved subformula. As you see, the rule is highly general, since it allows resolving on whole subformulas. Nevertheless, it will be used only resolution on atomic subformulas in this thesis. The proof of the soundness of the rule is similar to clausal resolution rule proof. Suppose the Interpretation I in which both premises are valid. In I, G is either true or false. If G (¬G) is true in I, so is F[G / ⊤] (F[G / ⊥]). From this point of view, it shows, that the resolution rule is nothing more that assertion of the type: If we have two formulas holding simultaneously and they contain the same formula, then we can deduce that either the common subformula is true in this interpretation then the truthfulness is assured by the first formula or the second formula in the opposite case. Now we can have a look to the question, how these facts influence the view of clausal resolution.

Consider following table showing various cases of resolution on the similar clauses.

Premise1	Premise2	Resolvent	Simplified	Comments
$a \vee b$	$b \vee c$	$(a \vee \bot) \vee (\top \vee c)$	$\top$	no sense
$a \vee \neg b$	$b \vee c$	$(a \vee \top) \vee (\top \vee c)$	$\top$	redundant
$a \vee b$	$\neg b \vee c$	$(a \vee \bot) \vee (\bot \vee c)$	$a \vee c$	right resolution
$a \vee \neg b$	$\neg b \vee c$	$(a \vee \top) \vee (\bot \vee c)$	$\top$	no sense

As you see the order of premises is important! When you want to make a reasonable resolvent you have to consider, which formula has to be taken as positive premise. In the clausal case, it is trivial question, it is the atom without negation. As you find, the non-clausal case will be also very simple.

Let's have a look into an example of a non-clausal refutation.

Example 3.1

(1) $a \wedge c \leftrightarrow b \wedge d$ **(axiom)**

(2) $a \wedge c$ **(axiom)**

(3) $\neg [b \wedge d]$ **(axiom) – negated goal**

(4) $[a \wedge \bot] \vee [a \wedge \top]$ **(resolvent from (2),(2) on c)** $\Rightarrow$
a

(5) $[a \wedge \bot] \vee [a \wedge \top \leftrightarrow b \wedge d]$ **((2),(1) on c)** $\Rightarrow$
$a \leftrightarrow b \wedge d$

(6) $\bot \vee [\top \leftrightarrow b \wedge d]$ **((4),(5) on a)** $\Rightarrow$
$b \wedge d$

(7) $\bot \wedge d \vee \top \wedge d$ **((6), (6) on a)** $\Rightarrow$
d

(8) $b \wedge \bot \vee b \wedge \top$ **((6), (6) on b)** $\Rightarrow$
b

(9) $\bot \vee \neg [\top \wedge d]$ **((8),(6) on b)** $\Rightarrow$
$\neg d$

(10) $\bot \vee \neg\top$ **((7),(9) on d)** $\Rightarrow \bot$ **(refutation)**

In the above example, you can see how simply it is to handle general formulas. Of course, something of used manipulations was not discussed (how to select formulas order to not produce redundant resolvents). Simplification used above is also not an essential need, but it was performed only for lucidity. It is eventual to retain the resolvents unsimplified until it is completely empty of atoms and then to determine logical value of the resolvent.

There is an important case of resolution called self-resolution describing resolution on one formula.

Definition 3.1: General self-resolution.

$$\frac{F[G]}{F[G / \bot] \vee F[G / \top]}$$

This type of rule allows us to perform "strange", but in some cases efficient, way of refutation the set of formulas as a whole formula. Again, consider the set from example 3.1.

Example 3.2

$a \wedge c \leftrightarrow b \wedge d$ **(axiom)**

$a \wedge c$ **(axiom)**

$\neg [b \wedge d]$ **(axiom) – negated goal**

Now we translate it to one formula:

(1) $[a \wedge c \leftrightarrow b \wedge d] \wedge [a \wedge c] \wedge \neg [b \wedge d]$

(2) $[\bot \wedge c \leftrightarrow b \wedge d] \wedge [\bot \wedge c] \wedge \neg [b \wedge d] \vee [\top \wedge c \leftrightarrow b \wedge d] \wedge [\top \wedge c] \wedge \neg [b \wedge d]$ (resolving on a) $\Rightarrow$

$[c \leftrightarrow b \wedge d] \wedge c \wedge \neg [b \wedge d]$

(3) $[\bot \leftrightarrow b \wedge d] \wedge \bot \wedge \neg [b \wedge d] \vee [\top \leftrightarrow b \wedge d] \wedge \top \wedge \neg [b \wedge d]$ (resolving on c) $\Rightarrow$ $[b \wedge d] \wedge \neg [b \wedge d]$

(4) $[\bot \wedge d] \wedge \neg [\bot \wedge d] \vee [\top \wedge d] \wedge \neg [\top \wedge d]$ (resolving on b) $\Rightarrow$

$d \wedge \neg d$

(5) $\bot \wedge \neg \bot \vee \top \wedge \neg \top$ (resolving on d) $\Rightarrow$

$\bot$ (refutation)

This type of resolution has two advantages as you saw in the example. It leads to the refutation quickly and without the need of deciding, if the resolvent will be redundant or not. Unfortunately, the self-resolution is not suitable for huge formulas and non-propositional instances.

3.2 Modifications.

The general resolution defined above is the base for refining other special cases. These modified versions are used in the application, in order to attain the best solving time for non-propositional cases. First modification resolves at one occurrence of the resolving subformula in the negative premise.

Definition 3.2: Partial General Resolution.

$$\frac{F[G] \quad F'[G]}{F[G / \bot] \vee F'[G \mid \top]}$$

In the Partial resolution in the negative premise, all the resolved subformulas remain with exception of one occurrence. Let's consider an example generated automatically by the GERDS application.

Example 3.3

Source formulas (axioms) :
F0 : ¬a∧¬b∧c∧d∨¬a∧¬b∧¬c∧d.
F1 (¬query) : ¬[¬a∧¬b].
Deduction by partial resolution:

R0 [F1&F0] : b∨¬a∧¬b∧¬c∧d. (resolves on a, but the second a from F0 retains in R0)
R1 [R0&F0] : ¬a∧¬c∧d∨¬a∧¬b∧¬c∧d.
R2 [R1&F1] : b.
R3 [R2&F0] : ¬a∧¬b∧¬c∧d.
[R3&R2] : YES. (refutation)

In this example, you can see resolvents in simplified form and processed by factoring rule. For the details about the results, see the section describing the programming of the application.

Another modification resolves only one occurrence of the subformula G in both premises.

Definition 3.3: Restricted General Resolution.

$$\frac{F[G]\quad F‘[G]}{F[G \mid \bot] \vee F’[G \mid \top]}$$

Example 3.4

Source formulas (axioms) :
F0 : ¬a∧¬b∧c∧d∨¬a∧¬b∧¬c∧d.
F1 (¬query) : ¬[¬a∧¬b].

R0 [F1&F0] : b∨¬a∧¬b∧¬c∧d.
R1 [R0&F1] : b.
R2 [R1&F0] : ¬a∧¬b∧¬c∧d.
R3 [R2&F1] : a.
R4 [R3&F0] : ¬a∧¬b∧c∧d.
[R4&R3] : YES.

Since the example is propositional, the next its general resolution deduction is shorter.
F0 : ¬a∧¬b∧c∧d∨¬a∧¬b∧¬c∧d.
F1 (¬query) : ¬[¬a∧¬b].

R0 [F1&F0] : b.
[R0&F0] : YES.

Are these refined rules sound? Consider the proof of the general resolution. Suppose interpretation I, in which both premises are valid. Now if G is true in I, then F[G | ⊤] is true in I, because substituted G has to be true in I, all other occurrences of G remains unchanged and these occurrences still remains true in I and it is not significant how many occurrences we substitute. Identically we can solve the contrary case (false). It can be also understood as an simpler instance of general resolution.

3.3 Polarity-Based Restrictions.

When we apply the inference rule to some premises, it is a natural question, how the resolvent arisen from them can influence the inference process. First, we have look in an approach presented in [Ba97]. Since it is a simple way to avoid the combinatorial explosion of resolvents, we will stop on it, though it is practically used another self-devised technique. Initially the notion of polarity is given.

Definition 3.4: **Polarity.**

A subformula F’ in E[F’] is said to be positive (resp. negative) if E[F’/⊤] (resp. E[F’/⊥) is a tautology. In that case F’ (resp. ¬F’) implies E.

For example, in a disjunction A ∨ B both A and B are positive, whereas in a conjunction A ∧ B the two subformulas A and B are neither positive nor negative. A subformula may occur both positively and negatively (e.g., A in A ∨ ¬A or A ↔ A), in which case the formula is said to be a tautology. The determining whether an atom A is positive or negative in E requires to check if E[A / ⊤] or E[A / ⊥]. It can be simply done by these criteria:

Theorem 3.1: Polarity criteria.

1. F is a positive subformula of F.
2. If $\neg G$ is a positive (resp. negative) subformula of F, then G is a negative (resp. positive) subformula of F.
3. If $G \vee H$ is a positive subformula of F, then G and H are both positive subformulas of F.
4. If $G \wedge H$ is a negative subformula of F, then G and H are both negative subformulas of F.
5. If $G \rightarrow H$ is a positive subformula of F, then G is a negative subformula and H is a positive subformula of F.
6. If $G \rightarrow \bot$ is a negative subformula of F, then G is a positive subformula of F.

The proof of the theorem is trivial and it is established on the notoriously known sense of logical connectives. Now it is possible to state two restrictions based on above.

Theorem 3.2: Redundancy of general resolution.

An inference by general resolution is redundant if the negative premise contains a positive occurrence of the resolved atom or if the positive premise contains a negative occurrence of the resolved atom.

Proof: If the negative premise contains a positive occurrence of the resolved atom A, then the resolvent appears as follows: $F[A / \bot] \vee F'[A / \top] \Rightarrow F[A / \bot] \vee \top \Rightarrow \top$. If the positive premise contains a negative occurrence of the resolved atom A, then the resolvent appears as follows: $F[A / \bot] \vee F'[A / \top] \Rightarrow \top \vee F'[A / \bot] \Rightarrow \top$. In both these instances resolvents degenerate to tautologies. In the refutational proof, such cases are improductive, i.e. from these resolvents can't be deduced false.

Theorem 3.3: Redundancy of general self-resolution.

An inference by general self-resolution is redundant if the resolved atom occurs positively or negatively in the premise.

Proof: If the premise contains a positive occurrence of the resolved atom A, then the resolvent appears as follows: $F[A / \bot] \vee F[A / \top] \Rightarrow F[A / \bot] \vee \top \Rightarrow \top$. If the premise contains a negative occurrence of the resolved atom A, then the resolvent appears as follows: $F[A / \bot] \vee F[A / \top] \Rightarrow \top \vee F[A / \bot] \Rightarrow \top$. In both these instances resolvents degenerate to tautologies. In the refutational proof, such cases are improductive, i.e. from these resolvents can't be deduced false.

Example 3.5

Let's consider two premises:

1. $\neg A$ – A is negative.
2. $A \wedge B$ – A is neither positive nor negative.

The resolvent of 1. and 2. is $\neg\bot \vee [\top \wedge B] \Rightarrow \top$.

3.4 Extended Polarity.

As it was noticed above, it is important to decide which of the two premises to be taken as positive. It has been developed a simple way to decide it during making of this thesis. It is an extended notion of polarity, which is similar to definition of Murray (1982), but the usage is different here.

Definition 3.5: Extended Polarity.

The subformula F is said to be positive (resp. negative) in E if after the transformation of E to conjunction-normal form, where F is treated as an atom, F would not be negated (would be negated).

It is clear, that every subformula has some polarity and if some of its superior connectives is equivalence, then it is both positive and negative. Following theorem gives algorithm for determination of polarity.

Theorem 3.4: Extended Polarity criteria.

1. F is a positive subformula of F.
2. If ¬G is a positive (resp. negative) subformula of F, then G is a negative (resp. positive) subformula of F.
3. If G ∨ H is a positive (resp. negative) subformula of F, then G and H are both positive (resp. negative) subformulas of F.
4. If G ∧ H is a positive (resp. negative) subformula of F, then G and H are both positive (resp. negative) subformulas of F.
5. If G → H is a positive (resp. negative) subformula of F, then G is a negative (resp. positive) subformula and H is a positive (resp. negative) subformula of F.
6. If G ↔ H is a subformula of F, then every subformula of G and H is positive and negative subformula of F.

Then it is possible <u>to set the formula with positive polarity as the positive premise</u>. This solves the problem of wrong order of premises i.e. it avoids redundant resolvents.

Example 3.6

Source formulas (axioms) :
F0 : a∨b.
F1 : ¬b∨c.
F2 (¬query) : ¬[a∨c].

R0 [F2&F2] : ¬c.
R1 [F2&F1] : ¬b.
R2 [F2&F0] : b.
R3 [F1&F2] : ¬b.
R4 [F1&F0] : c∨a. (c as negative premise)
R5 [F0&F2] : b.
R6 [F0&F1] : a∨c. (a as positive premise)
[R5&R3] : YES.

R4 was created as resolvent of F1 and F0 where F1 was treated as a negative premise: **(¬⊤** ∨ **c)** ∨ **(a** ∨ ⊥).

3.5 Simplification.

In the above subsections, it was applied the obvious notion of simplification for formulas. Although it is the clear process, the rewrite rules, which are the sources for the simplification, are stated here.

At the beginning, we mention the rules for eliminating logical constants from conjunctions, disjunctions and negations:

$$
\begin{array}{rcl@{\qquad}rcl}
A \land \bot & \Rightarrow & \bot & \bot \land A & \Rightarrow & \bot \\
A \land \top & \Rightarrow & A & \top \land A & \Rightarrow & A \\
A \lor \bot & \Rightarrow & A & \bot \lor A & \Rightarrow & A \\
A \lor \top & \Rightarrow & \top & \top \lor A & \Rightarrow & \top \\
\neg\bot & \Rightarrow & \top & \neg\top & \Rightarrow & \bot
\end{array}
$$

For other connectives, there are similar rules:

$$
\begin{array}{rcl@{\qquad}rcl}
A \rightarrow \bot & \Rightarrow & \neg A & \bot \rightarrow A & \Rightarrow & \top \\
A \rightarrow \top & \Rightarrow & \top & \top \rightarrow A & \Rightarrow & A \\
A \leftrightarrow \bot & \Rightarrow & \neg A & \bot \leftrightarrow A & \Rightarrow & \neg A \\
A \leftrightarrow \top & \Rightarrow & A & \top \leftrightarrow A & \Rightarrow & A
\end{array}
$$

Another important rule reducing the length of a formula is the factoring rule. The clausal form of the rule could be presented as follows:

$$\frac{a_1 \lor \; .. \lor a_n \lor a \lor a}{a_1 \lor \; .. \lor a_n \lor a}$$

where a_x and a are arbitrary atoms and the order of the atoms is insignificant.

This rule can be used also in the general case (general formulas) as a partial simplification technique.

3.6 Lifting of Inferences.

Before we start with explanation it should be stated that the following unification process doesn't allow an occurrence of the equivalence connective. It is needed to remove them by the following rewrite rule:

$$A \leftrightarrow B \Rightarrow [\, A \rightarrow B \,] \land [\, B \rightarrow A \,]$$

The general resolution presented was based on the propositional calculus. The lifting of resolution inferences to formulas with indiscriminated variables into universal and existential ones follows below.

For instance, general resolution.

$$\frac{F[G] \quad F'[G]}{F[G \,/\, \bot] \lor F'[G \,/\, \top]}$$

is lifted to

$$\frac{F[G_1,\ldots,G_k] \quad F'[G'_1,\ldots,G'_n]}{F\sigma[G \,/\, \bot] \lor F'\sigma[G \,/\, \top]}$$

where σ is the most general unifier (mgu) of the atoms $G_1,...,G_k$, $G'_1,...,G'_n$, $G = G_1\sigma$, and in contrast with [Ba97] it is not supposed renaming of the variables for the purposes of the application. We can suppose, that every variable occurring in a quantifier has its own identifier (for example memory address), which is assigned to variable occurrence. Technical details can be found in the section describing the programming of the application.

For this open problem it has been devised the extension of most general unifier. At first it is needed to introduce some supporting notions:

The following discrimination of existential and universal variables is needed for the Variable Unification Restriction definition:

When we speak about existential and universal variables, it is related to its notion with respect to the scope of the whole formula e.g. In $\exists X \forall Y\ p(X,Y) \rightarrow a(Z)$ Y variable to be treated as an existential variable, because the p(X,Y) subformula has negative extended polarity. It means, if we translate the formula into clausal form, Y would transform into existential variable. We define the discrimination as follows.

Definition: Discrimination of variables

Variable quantified by existential (resp. universal) quantifier is said to be globally existential (resp. globally universal), if the extended polarity of the subformula, which owns the quantifier (the quantifier prefixes it), is positive and it is said to be globally universal (resp. globally existential), if the the extended polarity is negative. If the polarity is both negative and positive, the variable is both globally existential and universal, but it only occurs if there is an equivalence connective in the formula. Since we required to remove this connectives by rewriting, there is not any problem for further definitions. If we speak about variables over scope of another variable, it means that the quantifiers of variables are located in superior nodes of the syntactical tree of the formula (in prenex form they prefixes the quantifier of mentioned variable).

Definition: Variable Unification Restriction

Variable Unification Restriction holds if one of the conditions i. or ii. is satisfied:

One of the terms is only a globally universal variable and the second one is not a globally existential variable. (non-existential case)

One of the terms is a globally universal variable, the second one is a globally existential variable, and every globally universal variable over the scope of the existential one has assigned some term.

This restriction performs the same job as skolem constants in clausal form. The unifiability of a gl. existential variable into a gl. universal variable is possible only if every gl. universal variable, which the existential variable depends on, has been substituted by a term already. Remark: The above definition determines, that two globally existential variables can't be unified, that is clear.

Basically, the mgu idea is following:

Definition: Most general unifier.

When both the terms, which have to be unified, are of the type string, number or functor without parameters then they are unifiable iff its type is the same (e.g. string and string and so on) and their identifiers match.

When one of the terms is a variable then it is unifiable with the second one iff Variable Unification Restriction holds. Then it is supposed that mgu substitutes the first variable to the second one and we do not require renaming, as it is obvious. If both the variables are globally universal then it is not significant, which is selected as the first one. If one of them is existential, we select it as the first.

If both the terms are of the type functor with arguments, then they are unifiable iff all the arguments are unifiable by the same procedure from the point i. (The order of unification to be mentioned with respect to unification of variables, so it tries to unify the atom until it expands the mgu from the first ununified term.)

There is no other possibility to unify two terms, except that two object of unification are the same physically (e.g. during the self-resolution the occurrence of two variables points to the same variable).

Let's have a look in an example of well-known facts:
It doesn't hold **∀X∃Y p(X, Y) ⊨ ∃Y∀X p(X,Y)** and
it holds **∃Y∀X p(X, Y) ⊨ ∀X∃Y p(X,Y).**
(General Y for all X can't be deduced from Y specific for X but contrary it holds.)

Source formulas (axioms) :
F0 : ∀X∃Y p(X, Y).
F1 (¬query) : ∀Y∃X ¬p(X, Y).

[F1&F1] : ⊥∨ ⊤.
[F0&F0] : ⊥∨ ⊤.

In this sample F0 and F1 can't resolve, since **∀X∃Y p(X, Y)** and **∀Y∃X ¬p(X, Y)** have no unifier. It is impossible to substitute universal X from F0 with X from F1, because X from F1 is existential and its superior variable Y is not assigned with a value. Counter-example with variable Y is the same instance and it is not allowed to substitute anything into an existential variable.

However, in the next example a unifier exists.
Source formulas (axioms) :
F0 : ∃Y∀X p(X, Y).
F1 (¬query) : ∃X∀Y ¬p(X, Y).

[F1&F0] : YES.
[F0&F1] : YES.

Hereinabove the universal variable X from F0 could be assigned with existential Y because Y in F1 has no superior variable. Then existential Y from F0 can substitute the universal Y from F1 for the same reason.

4 Resolution strategies.

4.1 Refutation.

As it has been written, the theorem proving method, which is used, is called refutational proof. Firstly the goal is negated, it is added to the set of axioms and then one searches for false formula using inference rules. Completeness of the refutational resolution proof is almost done by the proof for clausal instance presented in [Lu95]. It only should be replaced the conception of clauses by general formulas and consider the proof of soundness of the general resolution rule.

In next sections, several types of resolution strategies are presented, which may avoid generating huge amount of resolvents. In the beginning breadth-first and depth-first search are recalled, then important characteristics of linear search, filtration strategy and support-set strategy are summarized, in the end it is presented self-devised algorithm to reduce redundancy. Very good source of information about these strategies can be found in [Ma93].

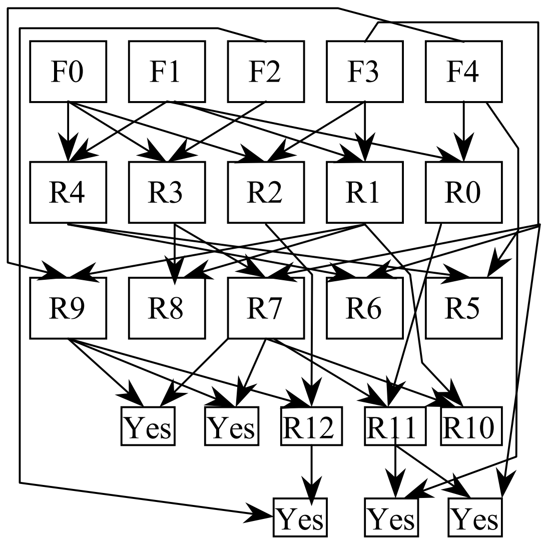
4.2 State Space Search.

Breadth-first search lies in generating all resolvents, which are possible to generate from the source set of formulas and they are called first-order resolvents. After it continues resolution of all the resolvents of the second order, which are resolved from at least one premise of first-order resolvents and so on. We can see such proof here:

Example 4.1

Source formulas (axioms) :
F0 : [a(X)∧g(X)→b(X)].
F1 : [b(X)∧g(X)→c(X)].
F2 : a(a).
F3 : g(a).
F4 (¬query) : ¬c(Y).

R0 [F4&F1] : ¬[b(Y)∧g(Y)].
R1 [F3&F1] : [b(a)→c(a)].
R2 [F3&F0] : [a(a)→b(a)].
R3 [F2&F0] : [g(a)→b(a)].
R4 [F1&F0] : [[g(X)→c(X)]∨¬[a(X)∧g(X)]].
R5 [R4&F4] : [¬g(X)∨¬[a(X)∧g(X)]].
R6 [R4&F3] : [c(a)∨¬a(a)].
R7 [R3&F3] : b(a).
R8 [R3&R1] : [¬g(a)∨c(a)].
R9 [R1&F4] : ¬b(a).
R10 [R1&R7] : c(a).
R11 [R0&R7] : ¬g(a).
[R11&F3] : YES.
Y = a.
[R10&F4] : YES.
Y = a.



[R9&R7] : YES.
Y = a.
R12 [R9&R2] : ¬a(a).
[R7&R9] : YES.
Y = a.
[R12&F2] : YES.
Y = a.

Figure 4.1. shows how the resolvents were obtained. You can see that there are four levels of resolvents.

Depth-first search generates resolvent from two premises of source set and then applies resolution on the result and other formula (resolvent or axiom) and then recursively until no resolvent can be generated. Then the algorithm returns to previous level (performs backtracking) and continues with another possible resolvent. This type of strategy is not complete i.e. it could stay in an infinite loop for inconsistent set and then no refutation is found, but it can lead to a false formula quicker than with breadth-first search. Next example is proved using linear strategy, which is an instance of depth-first search.

Example 4.2

Source formulas (axioms) :
F0 : [a(X)∧g(X)→b(X)].
F1 : [b(X)∧g(X)→c(X)].
F2 : a(a).
F3 : g(a).
F4 (¬query) : ¬c(Y).

R0 [F4&F1] : ¬[b(Y)∧g(Y)].
R1 [R0&F3] : ¬b(a).
R2 [R1&F0] : ¬[a(a)∧g(a)].
R3 [R2&F3] : ¬a(a).
[R3&F2] : YES.
Y = a.
R4 [R2&F2] : ¬g(a).
[R4&F3] : YES.
Y = a.

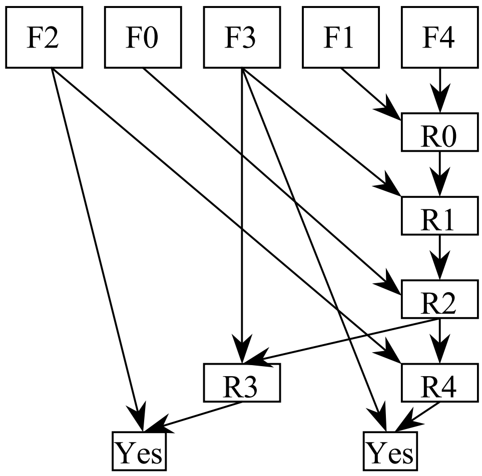


Figure 4.2. illustrate the simplicity of linear proof in comparison to previous example.

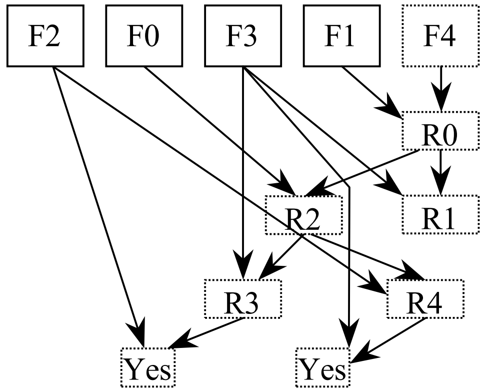
4.3 Common strategies.

Common strategies to reduce the set of resolvents include the wide-used Linear strategy. Example 4.2 was produced using this type of strategy. Linear strategy utilizes last generated clause as one of the premises. Linear strategies preserve the sequence of a proof. It is a base technique for logic programming. Other strategies are primarily intended for breadth-first search, but can be also invoked in depth-first search. Already mentioned disadvantage implies from its incompleteness. In the application, we use two different notions of linear search – linear, which generates resolvents only from goal and modified linear search, which is not restricted only to goal.

The Support Set Strategy is simple, but also incomplete strategy. It rises from the fact that there is an consistent subset in every source set of formulas. It is obvious that the resolvents from this subset can't lead to false formula. That's why this strategy allows to resolve such premises, from which one is the goal or its descendant. Let's again have a look in an example for the same set of formulas as above.

Example 4.3

R0 [F4&F1] : ¬[b(Y)∧g(Y)].
R1 [R0&F3] : ¬b(a).
R2 [R0&F0] : [¬g(Y)∨¬[a(Y)∧g(Y)]].
R3 [R2&F3] : ¬a(a).
R4 [R2&F2] : ¬g(a).
[R4&F3] : YES.
Y = a.
[R3&F2] : YES.
Y = a.



Support set strategy in this example is far more efficient than the unoptimized search.

The filtration strategy is the next resolvent reducing method. Two premises A,B can be resolved only if one of these conditions holds:

1. A or B is from the source set of formulas
2. A is a descendant of B or B is descendant of A.

(Descendant notion refers to the resolution tree.)

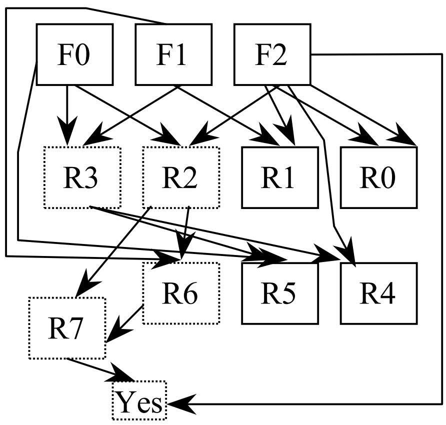
It is complete strategy, although it is not so efficient as last strategy.

Example 4.4

With filtration:

Source formulas (axioms) :
F0 : a→b∧g.
F1 : b∧g→c.
F2 (¬query) : ¬[a→c].

R0 [F2&F2] : ¬c.
R1 [F2&F1] : ¬[b∧g].
R2 [F2&F0] : b∧g.
R3 [F1&F0] : [g→c]∨¬a.
R4 [R3&F2] : ¬g∨¬a.
R5 [R3&F0] : c∨¬a.
R6 [R2&F1] : g→c.
R7 [R2&R6] : c.
[R7&F2] : YES.



4.4 Resolution strategies and redundant resolvents.

In this paragraph we will discuss the method which is the main point of the work on any automated prover. There is a lot of strategies which makes proofs more efficient when we use refutational proving, i.e. in proofs where it is added negated goal into the set of axioms and the empty formula (false) as a final resolvent detects a successful proof. We consider well-known strategies – filtration s., support set s. or their modifications. One of the most effective strategies is eliminating of consequent formulas. It means the check if a resolvent is not a consequence of a source formula or a previous resolvent. Then it is reasonable to not include the resolvent into the set of resolvents, because if the refutation can be deduced from it, then so it can be deduced from the older resolvent, which it imply of. The clausal case is trivial. Consider these two clauses p(X,a) and p(b,a) ∨ r(Y). It is clear that the second clause implies from the first (if we suppose that set of constant objects is closed – only b as the first argument in p predicate is considered). It has been devised an algorithm for the non-clausal case based on self-resolution rule. This algorithm highly reduces solving time, as it was proved in the application. Before it was introduced into the inference core, proofs were useless, due to their complexity. The method requires plenty of time; nevertheless the gain is high. If we want to express it simply: "Almost everything is better than to accept redundant resolvent". As we will see, there is such instance – infinite loop. The idea is quite simple and can be expressed in the following definition and theorem.

Definition: Consequent formula.

Formula F is a consequent formula in a refutational proof if there is a formula G in the set of resolvents or source formulas, where G → F holds.

Theorem: Detection of consequent formulas (DCF).

Formula F is a consequent formula of G if it is continually performed self-resolution on the formula ¬[G → F] until it has logical value and this logical value is false.

Proof: We can refer to a proof of completeness of the refutational resolution. When we have an inconsistent set of formulas, it is assured by the completeness that we reach a false formula. And since one formula forms a set and the self-resolution is a special case of general resolution, we can say that if ¬[G → F] is inconsistent then we prove it by self-resolution i.e. we prove that G → F is a tautology. We can argue, what happens if the formula is not a consequent formula. But from the properties of self-resolution it is clear, that the algorithm <u>must</u> finish with true or false in the propositional case, because self-resolution erases atoms in sequence until none retains (propositional logic is decidable). The non-propositional case is more complicated, but the solution to the eventuality, that the expression ¬[G → F] could never resolve to logical value, is clear. We may accept for exceptional instances that a consequent formula will be added to the set of resolvents rather than the possibility to end in an infinite loop. This is better alternative, so we can limit the theorem iterations by time or certain number. Then we assure that the theorem procedure is finite and in the worst case the redundant resolvent is added. It is the user responsibility to find proper limitation in the application. The experience with the prover says that such a restriction will not affect proofs almost anyhow.

The proof of completeness of the general resolution was not exactly done yet in this paper. Let's have a look at the <u>idea</u> of an original one. In the clausal case the proofs of completeness may vary, but we take the proof based on the number of excess literals presented in [2]. It is an inductive method. The parameter excess(S) = number of literals in S – number of clauses in S, where S is a set clauses. The completeness is stated as follows: If S is an unsatisfiable set of clauses, then there exists a refutation of S. Suppose that S is unsatisfiable.

For excess(S)=0: Either S consists of empty clause only or there are only literals in S. In this case there are 2 atoms mutually negative and we can resolve empty clause from them.

For excess(S)=n (n>0): S must contain clause C with more than one literal. S = S' + C. C = L ∨ C'. L is literal. {L ∨ C'} + S' is unsatisfiable, if (1) C' + S' and (2) L + S' is unsatisfiable. From induction hypothesis we have assured it is provable (1) and (2), because excess parameter is less than n. If we apply the proof for (1) to S we deduce either empty clause or L. If L is deduced we can continue with applying proof of (2) and we must deduce empty clause. So the induction is finished.

We can simply lift this proof for non-clausal instances. We suppose that the set of general formulas is unsatisfiable. It could be simply proved that every formula has its (logically) equivalent clausal representation. After the transformation into clausal rep. it should be discussed what form has the general resolution rule. With respect to complexity of formulas the general resolution represents one or more resolution steps in the clausal case. The excess(S)=0 case is the same, but if we have it assured for excess(S')=m < excess(S)=n, then it works in different way. The literal L may occur in more than one clause. So we must differentiate that S = S' + C1 + C2 + ... + Cn. The proof then must consist of a sequence in which we resolve every occurrence of L subsequently, after we use (1) deduction sequence on particular Ck.

Example:

Consider the formula **[a** ∨ **b]** ∧ **[¬b** ∨ **c]** and we prove that **[a** ∨ **c]** is a consequence of it.

¬[[a∨b]∧[¬b∨c]→a∨c]

¬[[⊥∨b]∧[¬b∨c]→ ⊥∨b]∨¬[[⊤∨b]∧[¬b∨c]→ ⊤∨b] ⇒ ⊥

It was used the factoring rule for the simplification on the line (2). It is clear that consequences are redundant in refutational sequence in contrast to direct proofs.

Example:

Formula **a∧b** is a consequence of **[a↔b]∧a.**

¬[[a↔b]∧a→a∧b]

[¬[[⊤↔b]∧ ⊤→ ⊤∧b]∨¬[[⊥↔b]∧ ⊥→ ⊥∧b]]∧[¬[[⊥↔b]∧ ⊥→ ⊥∧b]∨¬[[⊤↔b]∧ ⊤→ ⊤∧b]] ⇒ ⊥

Formula (2) divides into conjunction of two resolvents, where both the combinations of positive and negative premise are created.

The important aspect of the theorem DCF lies in its simple implementation into an automated theorem prover based on general resolution. The prover handles formulas in the form of syntactical tree. It is programmed a procedure performing general resolution with two formulas on an atom. This procedure is also used for the implementation of the theorem. A "virtual" tree is created from candidate and former resolvent (axiom) connected by negated implication. Then it remains to perform self-resolution on such formula until a logical value is obtained.

Let's compare the efficiency of standard strategies and the above-defined one. Consider following examples. It was also used modified notion of partially performed general resolution i.e. not every atom in a premise was replaced by a logical value. This modification may shorten proofs in some cases.

Rn means n-th resolvent and the expression in brackets represents premises of it.

F0 : a↔b∧g. F1 : b∧g↔c. F2 (¬query) : ¬[a↔c].
R0 [F2&F1] : [¬a∨¬[b∧g]]∧[a∨b∧g]. **R1 [R0&F2] : ¬[b∧g]∨c.**
R2 [R1&F0] : ¬g∨c∨¬a. **R3 [R2&F2] : ¬g∨¬a.**
R4 [R3&F2] : ¬g∨c. **R5 [R4&F0] : c∨¬a.**
R6 [R5&F2] : ¬a. **R7 [R6&F2] : c.**
R8 [R7&F1] : b∧g. **R9 [R8&F1] : g↔c.**
R10 [R9&F2] : [g∨a]∧[¬g∨¬a]. **R11 [R10&F1] : a∨[b↔c].**
R12 [R11&R6] : b↔c. **R13 [R12&F2] : [b∨a]∧[¬b∨¬a].**
R14 [R13&F0] : a∨[a↔g]. **R15 [R14&F2] : ¬g∨¬c.**
R16 [R15&F1] : ¬c. **R17 [R16&F2] : a.**
[R17&R6] : YES.
Solving time : 0.22 s.

None of standard above-mentioned strategies was able to limit proof but DCF was. Their proof sequence was cancelled after several seconds when it contained more than 300 resolvents.

Most of them were the same formulas or its clones. We can argue, if such formulas can't be avoided simply by comparison of their symbols. With respect to possible combinations of atoms it couldn't be a good strategy, because it does not work with interpretation of formulas, but DCF does.

F0 : a∧¬b∧c∧d∨a∧¬b∧¬c∧d∨¬a∧¬b∧c∧d∨a∧¬b∧¬c∧d∨a∧¬b∧c∧d.
F1 (¬query) : ¬[¬a∧¬b∨a∧¬b].

R0 [F1&F1] : b.
R1 [F1&F0] : b∨¬b∧c∧d∨¬b∧¬c∧d∨¬b∧¬c∧d∨¬b∧c∧d.
R2 [F0&F1] : ¬b∧c∧d∨b.
R3 [F0&F0] : ¬b∧c∧d∨¬b∧c∧d∨¬b∧¬c∧d∨¬b∧¬c∧d∨¬b∧c∧d.
[R3&F1] : YES.
Solving time : 0.05 s.
(without restriction)
R0 [F1&F1] : b.
R1 [F0&F0] : ¬b∧c∧d∨¬b∧c∧d∨¬b∧¬c∧d∨¬b∧¬c∧d∨¬b∧c∧d.
[R1&F1] : YES.
(with DCF)

F0 has the form of DNF, so it is not suitable for clausal representation and despite of this fact the proof is short.

Another method to shorten proofs is implemented and it follows from trivial property of self-resolution (SR) discussed within DCF. We can simply apply SR to the resolvent itself and then we can decide if the formula is contradictory or logically valid. In the same way it is needed to limit such algorithm by some way, due to partial decidability of PL. The example of this method is presented in the first one of the section 4. When the reference to this method is marked, it is used notion of SR-*check*. But the SR-check also refers to the one step of the theorem DCF application.

5 Algorithms and programming interface.

5.1 Programming tools.

It was exploited excellent development environment for the production of the application performing theoretically presented manipulations in the previous chapters. The Borland Delphi 2 with the Object Pascal language is thought as this tool offering many objective extensions to standard Pascal. This language is utilized to present the algorithms for the inference techniques. It is supposed that the reader has an essential knowledge of Pascal. First we start with the data structures, the construction of parse trees and then we focus to resolution methods. To get an introduction to Pascal language see [Ji88] or electronic help of Delphi 2. There is a very good source of information about data structures such as stack in [Be71].

5.2 Data structures.

The General resolution deductive system has the frame for every logic program, which may contain source formulas and goals. It is divided to two windows – for the program and for results of inference. Whole frame is represented by TEditForm class and it encapsulates Editor for program and Output for results, both TMemo objects. We omit these objects, since they have no meaning for inference process and they are only visual components. We start with one member of TEditForm representing internally the logic program. It is the TPLProgram class.

```
TPLProgram = class
  Owner : TEditForm;
  ListF : TList;  { List of source formulas and queries. }
  ListR : TList;  { List of resolvents. }
  MGUn1  : TList;  { Temporal store for unification results. }
  Err : ErrorType; { Temporal store for compilation error. }
  Localpos : Longint; { Position of an error. }
  Strategy : TObject; { Type of the resolution strategy. }
  constructor Create(ow : TEditForm);
  destructor Destroy; override;
  procedure Generate; { Generates set of compiled formulas. }
  procedure XComp(var infix : PChar; var F : TFALFormula);
    { Compilation core. }
  procedure ClearFormulas;
  procedure PrintFormulas;
  procedure ClearResolvents;
  procedure Consist; { Encapsulates consistency of the ListF checking. }
end;
```

Here is visible the structure of the class. Key objects are ListF and ListR. They contain source formulas objects (resp. resolvents) represented by a parse tree of the class TFALFormula.

```
TFALFormula = class
   Cont : TSub; { Contents of the formula. }
   Owner : TPLProgram;
   Parent1 : TFALFormula;
   Parent2 : TFALFormula;
     { Parents of the formula in the meaning of resolution rule. }
   MF, ML : TAtom; { First and last atom. }
   Support : boolean;  { Indicator for the Support Set Strategy. }
```

```
    ...
  end;
```

Cont stores a reference to a subformula of the class TSub.

```
  TSub = class
    Neg : boolean;  { Flag of logical negation. }
    Ev : shortint;  { Indicator of the subformula logical value. }
    Pol : shortint; { Stores the polarity of the node. }
    L : TSub;
    R : TSub;        { Left and right subtree. }
    Ac : TObject;    { Parent object (logical connective). }
    Q : TQuant;      { Quantifier containing variables at this level. }
    ...
  end;
```

TSub is the base class for other descendants – TCon, TDis, TImp, TEqv and TAtom. They represent particular logical connectives. Every subformula has a reference to left and right subformulas. Ev may have three values: -1 – false, 1 – true, 0 – subformula is not evaluated. Ac stores pointer to the superior node (the root of a formula points to TFALFormula object). TAtom has special member indeed.

```
  TAtom = class(TSub)
      Id : TId; { Identifier of a formula. }
      ...
  end;
```

Id points to a Id term – predicate name. L and R inherited from the TSub object have different function here. It points to the next atom object (in the infix notation) instead of descendant logical object (TAtom has no logical descendant). TQuant class is a descendant of TList class. It contains list of variables quantified in a quantifier or additionally quantified free variables. TId has several descendants:

```
  TVariable = class(TId)
   Id : TId; { Substitution of the variable by an expression. }
   Ex : char;  { Indicates quantification of a variable. }
   New : TVariable; { Temporal pointer used in copying of the variable. }
   NewId : TId; { Temporal pointer determining substitution. }
   Used : boolean; { Flag of variable usage in expressions. }
   UnSub : boolean; { Flag determining substitution. }
   Ap : Boolean; { Used during unification. }
   Master : TSub; { Master subformula of the variable. }
   Watch : TVariable; { Link to goal requested source variable. }
   Name : string; { Logical name of the variable.}
  end;
```

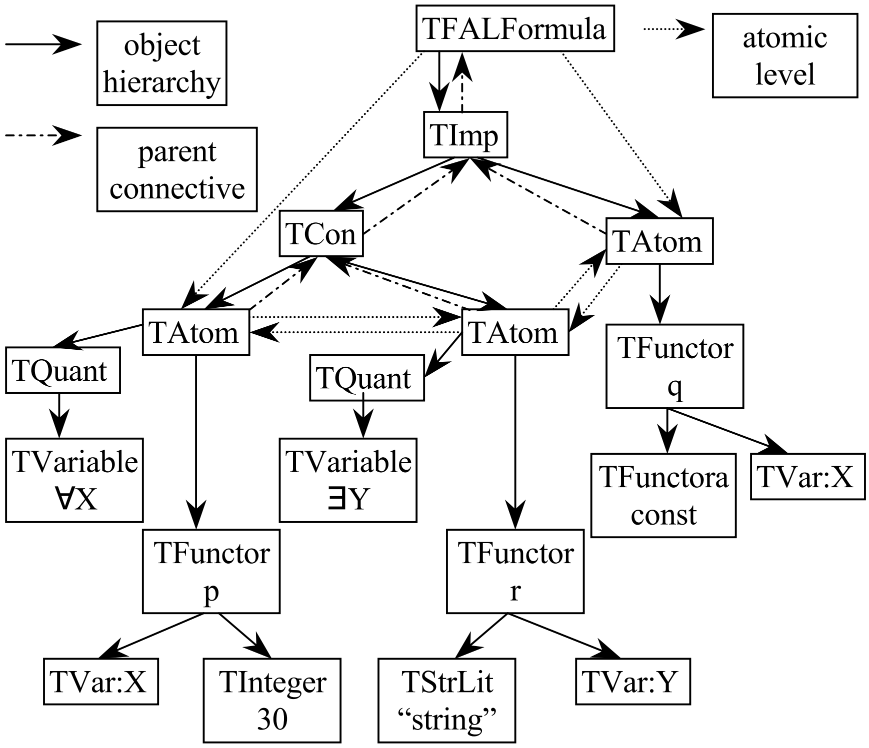
TVariable is the class representing certain variable quantified with a quantifier or quantified implicitly as a free variable. Id refers to a term substituted to the variable. Ex may obtain two values: cforall – constant of universal quantifier character $\forall$ or cexists – existential character $\exists$. New is used, when a new copy of a formula is created. It points to newly created variable. NewId points to a term, which has to be substituted and from this term is created a copy assigned later to New variable. Master refers to the subformula, which quantifies the variable. Watch is the observing reference to the variable from the goal of the logic program. It is created, in order to show the assignment of variables, which led to a contradiction. After direct compilation, all occurrences of a variable in terms are not assigned to one object of the class TVariable, but to another descendant of TId – TVar. TVar consist of Name only and serves as temporal class, before the TVariable is assigned to an occurrence.

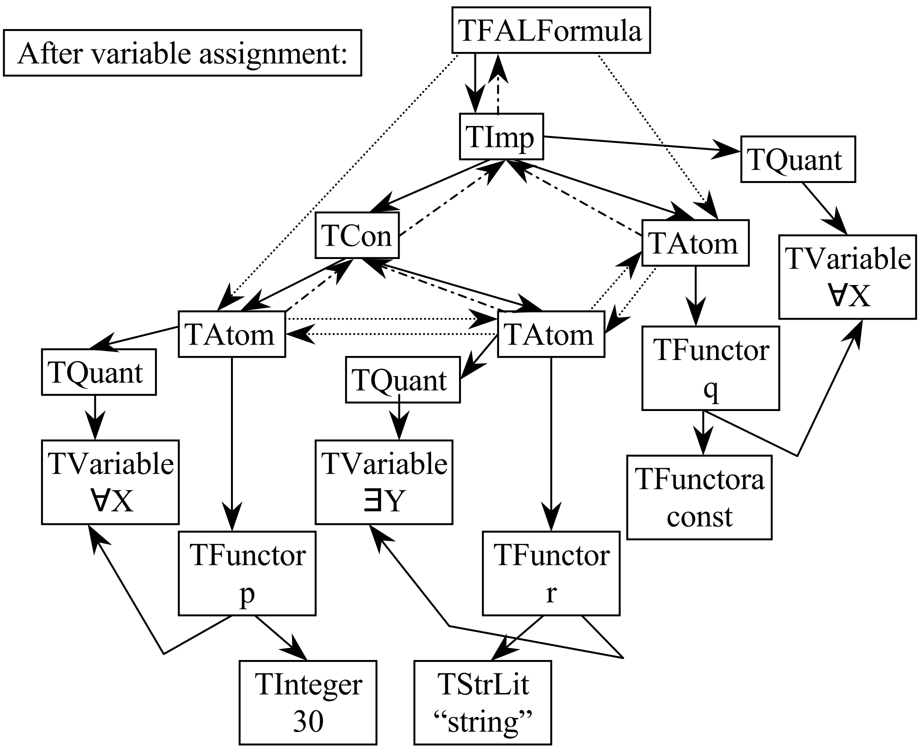
Further TId descendants are: TStrLit, TReal, TInteger, TFunctor and TFunctora, which represent string, real number, integer, functor with arguments and functor without arguments.

Now let's see into some illustrations of parse trees of formulas.
Consider this simple formula:

Example 5.1

∀X p(X,30) ∧ ∃Y r("string",Y) → q(const,X).
Direct compilation:





The example of the object hierarchy of the parse tree shows, how the variables are assigned to particular occurrences of them. After variable assignment there is a new quantifier, which quantifies former free variable X. It was created in the root of the tree, since free variables have global scope.

5.3 Parser.

Last subsection analyzed the data structures of formulas and indicated that the parse tree plays the main role in the practical use of general resolution. The editor of the logic program source is accessible as a null-terminated string and the main parsing procedure TPLProgram.XComp uses this string.

```
procedure TPLProgram.XComp;
 begin
  ...
  StackInit;
  cpos := 0; dontcare := 0;Error := OK;Getchar;
  if (ch = '?')and(infix[cpos] = '-') then
  begin
   query := true;GetCharI; GetChar;
  end;
  if Error = OK then Eqv;  { Go to parser / generator. }
  ...
  if Error = endcomp then
   begin
     Error := OK;sub := TSub(VPSStack^.Node);dispose(VPSStack);
     if query then F := TQuery.Create(self)
     else F := TFALFormula.Create(self);
     F.Cont := sub;sub.Ac := F;F.MF := FirstMol;F.ML := LastMol;
     F.PostComp;GetChar;
     if ch <> #0 then LocalPos := cpos-1 else LocalPos := cpos;
   end
  else
    begin
     if VPSStack <> nil then DestroyNode(VPSStack^.node);
     DisposeStack;LocalPos := cpos-inum-1;
    end;
 err := error;
 case error of
  ...
 end;
end;
```

Let's specify particular steps of the algorithm. At the beginning there are some initializations – StackInit initializes the stack, where objects as atoms and terms are stored until they are assigned to their parent objects, parse variable has reference to logic program string, LastMol and FirstMol are used for right assignment of first and last atom of a formula after successful compilation. GetChar (resp. GetCharI) gets into ch variable one lexical element from the editor ignoring blank characters (resp. inclusive blank chars). Then it decides if the formula is a goal. Then follows recursive parsing, which we will discuss in detail hereinafter. At the end the procedure checks for errors during compilation and if everything is right then the root of the parse tree is extracted from the stack and the appropriate TFALFormula structure is generated else the error message occurs.

Let's see the recursive parse algorithm (well explained in [Dv92]). Recursive parsing lies on programming procedures exactly by the Backus-Naur form, where every non-terminal has its own procedure and there is GetChar calling before a terminal and particular procedure calling for a non-terminal inside the procedure. Iteration and branch on condition are solved identically in

BNF and algorithmical level. Of course, the requirement of LL(k) language defined by BNF is required i.e. decision which case to choose in a non-deterministic instance must be found by looking ahead k characters. In this instance, the LL(1) language is presented in the second chapter.

```
procedure Eqv;
begin
 if Error = OK then
   begin
    Imp;
    while (ch = ceqv)and(error = OK) do
      begin
       Getchar;
       Imp;
       if error = OK then AssignSub(TEqv.Create);
      end;
   end;
end;
```

Here it is presented the procedure Eqv related to <Formula> non-terminal (expresses equivalence level with the lowest priority). It dives to Imp non-terminal procedure first and then continues zero or more times with the same non-terminal depending if some equivalence character is found. Every such equivalence is created and it has assigned L and R descendant by AssignSub procedure by the following way.

```
procedure AssignSub(Ob : TObject);   { Assigns formula its subtrees. }
begin
  Put(Ob);pom := Cut;
  TSub(pom^.node).R := TSub(VPSStack^.node);
  TSub(VPSStack^.node).Ac := pom^.node;
  garbage := Cut;dispose(garbage);
  TSub(pom^.node).L := TSub(VPSStack^.node);
  TSub(VPSStack^.node).Ac := pom^.node;
  garbage := Cut;dispose(garbage);SInsert(pom);
end;
```

Put procedure creates a stack object, which encapsulates every object's life in the stack and enables the logical object to be bounded in the stack with the succeeding object. Then it pops from stack with Cut procedure and it is stored in the pom variable. It is assigned the first object in the stack into the R pointer of pom object. Then the same is done with the second object. They are simultaneously removed from the stack since they are now elements of the parse tree. Resulting object is then put in the stack and waits until it is attached to its superior node.

The procedure ICE related to <Subformula> fully demonstrates the recursive character of such algorithm.

```
procedure ICE;
begin
if error = OK then
 begin
  case ch of
   cneg : begin
        Getchar; ICE;NegateF;
       end;
   cforall, cexists :
       begin
         Quant; ICE;
         if Error = OK then AssignQtoSub;
       end;
```

```
  'a'..'z', 'A'..'Z', '_', '0'..'9', '"', '(', '+', '-' :
      begin
        Atom;
      end;
   '[' : begin
        GetChar; Eqv;
        if (ch <> ']')and(error = OK) then Error := missbra;
        Getchar;
      end
  else if error = OK then Error := missexp;
  end;
 end;
end;
```

The cneg case refers to a negated subformula and it recursively calls the same procedure. The second case of quantifiers does the similar work.

The third case handles an occurrence of atom and it remains to solve the case of subformula enclosed in brackets. It is obvious that the other parser procedures are omitted, because they use the same principle.

5.4 Postprocessing.

There is still something undone after successful compilation. It is inevitable to assign variables instead of TVar objects for every occurrence of a variable, it is also needed to evaluate the extended polarity of nodes and it is useful to perform evaluation of constants connected by infix operators. These operations are encapsulated in the next procedure.

```
procedure TFALFormula.PostComp;
var TT1 : TAtom;
begin
  LastSub := nil;
  Dive('4', VarReAssign, [nil]); { Assignment of variables. }
  Dive('3', EvalBuilt, [nil]);  { Evaluation of infix operators. }
  Cont.EvalPol;
end;
```

Dive procedure has three arguments. It is the general procedure, which helps to browse parse tree without the need to write the same code for different actions. The first argument determines the manner of browsing inside the tree e.g. ‘3’ – postorder (first go to subtrees and then performs a operation on the node, details in [Be71]). The second argument sends a reference to the procedure, which does the actions. Last argument may send some additional information in the form of variant array i.e. special item of Object Pascal language, which allows to work with variable number of arguments of changeable types. The executive procedure (second par.) must have this form:

```
TDoProc =
  function(o : TObject; var tp : char; argums : array of const) : TObject;
    { Template for functions used as executive procedures
      for the recursive browsing of the syntactical tree.(ST) }
```

It accepts reference o to the current object, the currently valid mode of browsing and already mentioned variant arguments and it returns reference to an object, which will replace the assignment of the current node if the modification is needed.

Before we continue with the postprocessing, let’s have a look into an example of the implementation of the Dive procedure:

```
function TSub.Dive;
var i : longint;
```

```
begin
  case tp of
  '1' :
    begin
      Proc(self, tp, argums);
      if Q <> nil then  Q.Dive(tp, proc, argums);
      if L <> nil then  L.Dive(tp, proc, argums);
      if R <> nil then  R.Dive(tp, proc, argums);
    end;
  ...
    'r' :
    begin
      Result := TSub(ClassType.Create);
      Result.Neg := Neg; Result.Ev := Ev;Result.Pol := Pol;
      Result.Ac := Ancs; LastSub := Result;
      if Q <> nil then
      begin
        Result.Q := Q.Dive(tp, proc, argums);
        if Result.Q <> nil then
        with Result.Q do
        for i := 0 to Count-1 do
          TVariable(Items[i]).Master := Result;
      end;
      Ancs := Result; LastSub := Result;
      if L <> nil then Result.L := L.Dive(tp, proc, argums);
      Ancs := Result; LastSub := Result;
      if R <> nil then Result.R := R.Dive(tp, proc, argums);
      LastSub := Result;
      if Q <> nil then Q.Dive('q', proc, argums);
    end;
    ...
end;
```

Of course, a lot of code was cut short, but there still remains illustrative sample. The ‘1’ mode means preorder type of browsing i.e. First perform Proc (executive procedure) on the node and then go recursively into descendant trees. The ‘r’ mode expresses the action of copying the tree and as you see, the result variable assures the right assignment of new object members.

If we return to the beginning of this subsection, we can continue with explanation of the PostComp procedure.

```
{ Procedure for assignment of complex variable object for every occurrence of
  the variable. }
function VarReAssign(o1 : TObject; var tp : char; argums : array of const)
                                                   : TObject;
var S1 : TSub; i : integer; O : TVariable;
begin
  Result := o1;
  if o1 is TSub then
  ...
  else if (o1 is TVar) then
  begin
  S1 := LastSub; O := nil;
  if S1.Q <> nil then
    with S1.Q do
    begin
      for i := 0 to Count-1 do      { Searches for right TVariable object. }
       if TVariable(Items[i]).Name = TVar(o1).Id
       then O:=TVariable(Items[i]);
    end;
```

```
    while not(S1.Ac is TFALFormula)and(O = nil) do
    begin
      S1 := TSub(S1.Ac);
      if S1.Q <> nil then
      with S1.Q do
      begin
        for i := 0 to Count-1 do
          if TVariable(Items[i]).Name = TVar(o1).Id then
                              O := TVariable(Items[i]);
      end;
    end;
    if O <> nil then
    begin
      o1.Free;
    end
    else
    begin
      if S1.Q = nil then S1.Q := TQuant.Create;
      O := TVariable.Create(TVar(o1));
      O.Ap := false;
      O.Master := S1;   { If not found, creates new variable on  }
      o1.Free;          {  the top level.                        }
      O.Ex := cforall;S1.Q.Add(O);
    end;
    Result := O;
  end;
end;
```

It is important factor, which subformula is the nearest superior node in this executive procedure. It can be found from the LastSub variable. From this points it starts to search for the nearest variable, which match the name of the occurrence. It continues to the parent node and tries to match the name, but if it doesn't succeed until the root is reached, it creates a new variable and appends it into the root quantifier. It set up the Master property of the variable, which points to the subformula, which includes the quantifier encapsulating the variable.

```
  { Evaluates infix operators. }
function EvalBuilt(o : TObject; var tp : char; argums : array of const)
                                              : TObject;
var r : extended; N0, N1 : TObject; fc : char;
begin
  Result := o;
  if o is TFunctor then
  begin
    if (TFunctor(o).Functor[1] in infixoper) then
    begin
     N0 := (TFunctor(o).Params.Items[0]);
     N1 := (TFunctor(o).Params.Items[1]);
     while N0 is TVariable do
       N0 := TVariable(N0).Id;
     while N1 is TVariable do
       N1 := TVariable(N1).Id;
     if (N0 is TNumber)and(N1 is TNumber) then
     begin
       try
       case TFunctor(o).Functor[1] of
       '+' : r := TNumber(N0).GetNumber +
              TNumber(N1).GetNumber;
       '-' : r := TNumber(N0).GetNumber -
              TNumber(N1).GetNumber;
       '*' : r := TNumber(N0).GetNumber *
```

```
          TNumber(N1).GetNumber;
      '/' : r := TNumber(N0).GetNumber /
          TNumber(N1).GetNumber;
      end;
      except
        on EMathError do Exit;
      end;
      TFunctor(o).RFree;
      Result := TReal.CreateN(r);
    end;
   end;
  end;
end;
```

If the functor object is the current, it checks for infix operator and if both first and second argument is a number. In that case the functor is replaced by the resulting number object.

```
procedure TSub.EvalPol;
begin
  if Ac is TFALFormula then if Neg then Pol := -1 else Pol := 1
  else
  begin
    if ((Ac is TImp)and(TSub(Ac).L = self)) then Pol := -TSub(Ac).Pol
    else Pol := TSub(Ac).Pol;
    if Neg then Pol := -Pol;
  end;
  if (self is TEqv)or(self is TAtom) then Exit
  else { Equivalence causes zero priority. }
  begin
    if L <> nil then L.EvalPol;
    if R <> nil then R.EvalPol;
  end;
end;
```

The above procedure evaluates the (extended) polarity of the node. It utilizes the criteria from theorem 3.4. The first line of the procedure checks if it is the root of the parse tree and if so then it assigns polarity depending on the neg flag of the subformula. Then it copies the polarity from the superior node or if it is an antecedent of an implication, it copies the inverse value.

5.5 Theorem proving.

When one calls any type of proof of some source set of formulas with goals, the Generate procedure of TPLProgram object is called.

```
procedure TPLProgram.Generate;
var p1, p2 : longint; temp : PChar; Form, Form2 : TFALFormula;
    t1, t2 : double; infstr : string;
begin
  p1 := 0; p2 := 0; temp := PChar(Owner.UpdateCont);
  CurProg := self;ClearFormulas;
  ...
  while temp[0] <> #0  do        { Compiles until end is found. }
  begin
    Application.ProcessMessages;if stopf then Exit;
    Form := nil;XComp(temp, Form);
    if Form <> nil then
    begin
      Form.Simplify;
```

```
      if Form is TQuery then
      begin
        TQuery(Form).EvalQuery;   { Evals if it is query. }
      end
        else ListF.Add(Form);     { Or adds to ListF. }
      p1 := LocalPos;temp := @(temp[p1]);p2 := p2+p1;
    end
    else begin
      Owner.Editor.SelStart := p2+LocalPos+1;
      ClearFormulas; temp := PChar('');
    end;
  end;
  ...
end;
```

The temp variable points to the editor's text and it is passed through until no formula is generated. It may raise both by reaching the end of logic program or by a parse error. The XComp procedure returns the position directly after dot and blank characters terminating last formula. Source formulas are added to ListF container. If the generated formula is a query, then Generate calls its EvalQuery procedure, which is responsible for the proof. After solution of a goal, next query can be solved or succeeding source formulas may be compiled.

```
procedure TQuery.EvalQuery;
begin
  NegQ;Owner.ListF.Add(self);
  Support := true;
  Dive('v', nil, [nil]); { Marks variables requested by a query. }
  Owner.Consist;
  Owner.ListF.Remove(self);
  Free;
end;
```

At first the query is negated and it is added to ListF and its Support-set flag is set up. Mode 'v' for Dive procedure cause setting the Watch property of every variable in the goal to itself. It allows to all copies of a variable to sustain the reference to original query variable and it can be printed out after successful refutation. The TPLProgram procedure Consist includes several type of state-space search. Because it is a huge procedure, we will show only its part presenting linear strategy and we will also follow with linear strategy subprocedures.

```
procedure TPLProgram.Consist;
...
begin
  ...
{ Linear strategy. }
  if Strategy = Owner.Linear then
  begin
    rsv1 := 'F'+IntToStr(ListF.Count-1);
    TFALFormula(ListF.Items[ListF.Count-1]).AllPosRes4;
  end
  else
  ...
end;
```

Consist procedure calls AllPosRes procedure of the goal (last formula in the source set). AllPosRes tries to perform resolution rule on every formula of the source set and originated formulas. The procedure, which has to carry out the resolution of two formulas, is from the family of TryToRes procedures. Its function lies on passing through atoms of formulas and trying to find an unifier of them. If two atoms are unifiable, it checks for polarity and selects positive atom as positive premise (by setting up the premise flag). Then it calls Resolve procedure, which produces the resolvent. If both the atoms have zero polarity, it is suitable to generate two resolvents (one with positive premise of first formula and second with positive premise of second formula) or to generate conjunction of such resolvents. The detail listing of TryToRes is improductive due to its length. There is a lot of variations and optimizations indicated in the theoretical sections. At first it inspect formulas eligibility for resolution (filtration, support-set) and it may also generate more than one resolvent from a couple of formulas (on different atoms). After resolution, the resolvent is simplified and it remits to check of consequence (equivalence), if needed. When the resolvent is generated and it is not redundant or logical value, the linear strategy continues and applies the AllPosRes to the resolvent. Let's see to Resolve procedure.

```
function TFALFormula.Resolve;
...
 if premise then tr1 := -1 else tr1 := 1;
 if T1.neg then T1.Ev := -tr1 else T1.Ev := tr1;
 if self <> X then if T2.neg then T2.Ev := tr1 else T2.Ev := -tr1;
 { Resolves two matching atoms}
 if (getAt)or(Owner.Owner.GreatCut1.Checked) then
 { In the case of optimization,
 it searches for all other possible matching atoms. }
 begin
   TT1 := T1;polar1 := false;
   if (TT1 <> nil) then
   begin
      TT2 := MF; rs := 0;
      while (TT2 <> nil) do
      begin
        rs := TT1.UnifySL(TT2);
        if rs = 0 then TT2 := TAtom(TT2.R)
        else
        begin
          if polar1 then TT2.Ev := -tr1 else TT2.Ev := tr1;
          if TT2.neg then TT2.Ev := -TT2.Ev;rs := 0;TT2 := TAtom(TT2.R);
        end;
      end;
   end;
   if (self <> X)and
     ((getat)or(Owner.Owner.General.Checked)) then
   begin
      ... { Identical code, but for the second formula }
   end;
 end;
 PrepareMgu;FL.Cont := F3; F3.Ac := FL;Ancs := F3; TN2 := T2;
 F1 := Cont.Dive('r', nil, [nil]);   { Makes a copy of the first formula
                                       with substituted variables. }
 Ancs := F3; TM1 := LastMol; TN2 := nil;
 if (self = X)   then
 begin
   invert := true;F2 := X.Cont.Dive('r', nil, [nil]);invert := false;
 end
 else F2 := X.Cont.Dive('r', nil, [nil]);
 ...
end;
```

First mentioned lines perform assignment of a logical value to resolved atoms. It depends on premise flag, which indicate that the first formula – self has to be treated as the positive premise and the second formula X has to be regarded as the negative premise. Of course, it is useless to evaluate the second atom, if self=X. Next it evaluates all atoms from the positive premise, if the user requires it by checking off the General cut (Partial resolution) item from the application menu. If the general cut item is checked off, then all atoms from the second formula are evaluated and resolve procedure carries on general resolution. Otherwise it evaluates only two atoms and this is an implementation of restricted resolution. PrepareMgu assigns unifier substitution from the temporal TList store named mgu1 into the appropriate variable NewId property. Then new disjunction is created and copies of premises made by ‘r’ mode of the Dive procedure are assigned as L and R references to this disjunction. There is also special procedure for self-resolution called ResolveAWC.

5.6 Unification.

The key factor of lifting inferences to non-ground cases of formulas is the most general unifier. The function responsible for unification is TAtom.UnifyS and TAtom.UnifyId is its executive procedure, which unifies Id properties of two atoms.

```
function TId.UnifyId;
var varn : integer;
begin
  varn := mgu1.Count-1;
  repeat
    unres1 := 1;  { Tries to unify id terms until, it is clear that }
    unres := 1;   { no more changes are done.                       }
    varn := mgu1.count;
    Dive('5', IdUnif, [X]);
    if unres1 = 0 then unres := 0;
  until  (varn = mgu1.count)or(unres <> 0);
end;
```

The ground or non-existential instance is trivial and one pass is enough, but in the existential case the order of arguments in functors is important. That’s why it tries to unify two terms until any substitution is not realized or the unification is completely impossible. What it means? The unres variable is set to zero, if two constant objects can’t be unified, while unres1 is set zero even if some variable can’t be unified. So if a variable was taken in a wrong order, we can still remain it ununified and wait for the next pass, which will occur only if some other variable was unified and it has sense to examine if the last unified variable could be unified under changed conditions. The dive procedure serves also unification as the mode ‘5’ with IdUnif procedure.

```
function IdUnif(o : TObject; var tp : char; argums : array of const) : TObject;
...
begin
  Result := o;
  ...
  if (o is TFunctor) then
    ... { All arguments of a functor have to be unified.}
    else if (o is TFunctora) then
    begin
      if not((argums[0].VObject is TFunctora)and
        (TFunctora(argums[0].VObject).Functor = TFunctora(o).Functor))
        then unres := 0;
    end
    ...
    else if (o is TVariable) then
    begin
      begin
        if not(CurProg.Owner.Quant1.Checked)or(IsHigherObject(o, argums[0].VObject)) then
        begin
          mgu1.Add(o);mgu1.Add(argums[0].VObject); unres := 1;
```

```
        end
        else begin unres1 := 0; end;
      ...
```

Here it is presented the unification algorithm from section 3.6. The special attention is given to variable unification. It depends on IsHigherObject. This function returns true, if Variable Unification Restriction holds.

5.7 Simplification and Check of consequence.

In order to reduce the formula length, the simplification rules described above are used. This is an implementation of these rules.

```
function FSimplifyH(o : TObject; var tp : char; argums : array of const)
                                                  : TObject;
...
begin
  if (TSub(o).L <> nil)and(TSub(o).R <> nil)
    and((TSub(o).L.Ev <> 0)or(TSub(o).R.Ev <> 0)) then
  begin
    inc(numSimX);x1 := TSub(o).L.Ev; x2 := TSub(o).R.Ev;
    if o is TCon then xa := Min(x1, x2)
    else if o is TDis then xa := Max(x1, x2)
    else if o is TImp then xa := Max(-x1, x2)
    else if o is TEqv then xa := Min(Max(-x1, x2), Max(x1, -x2));
    if (xa <> 0) then
      begin
        TSub(o).L.RFree; TSub(o).L := nil;
        TSub(o).R.RFree; TSub(o).R := nil;TSub(o).Ev := xa;
        { The case, that leads to complete evaluation of a node
          e.g. (x and true) leads to x .}
      end
    else if (xa = 0) then
      begin
        negf := 0;
        if ((o is TImp)and(x2 = -1)) then negf := 1
        else  if ((o is TEqv)and((x1 = -1)or(x2 = -1))) then negf := 2;
        if x1 = 0 then o := TSub(o).L.SubstTo else o := TSub(o).R.SubstTo;
        if negf <> 0 then TSub(o).negate;
        { The case, that leads to a partial evaluation of a node.
          e.g. (x and false) leads to false .}
      end;
    if TSub(o).neg then TSub(o).Ev := -TSub(o).Ev;
  end;
  Result := o;
end;
```

At first it is evaluated the logical value of the subformula o depending on the connective based on (only for these purposes) three values- false, unknown, true. Then it is decided, whether the subformula will be completely removed or it will be removed only its descendant.

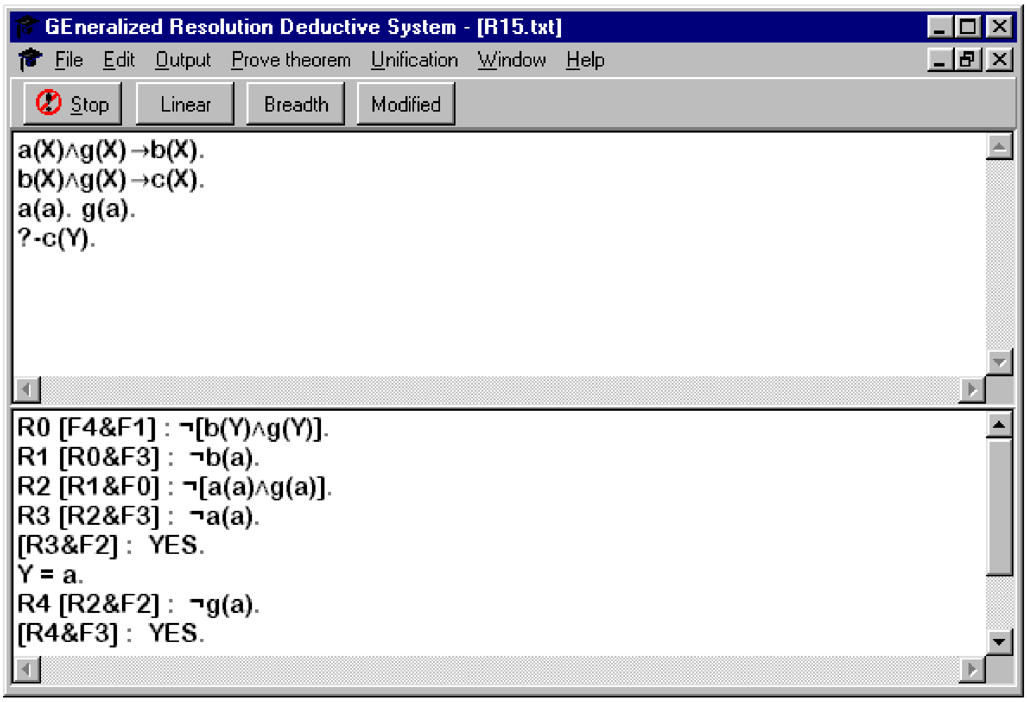
The decision about the redundancy implements the InsertToR procedure.

At first it stores the result of self-resolving on the examined formula. If a logical value is obtained, check of consequence has no sense. Otherwise it performs the following actions for every source formula and resolvent until the redundancy is proved. It creates virtual formula with negated implication in the root, L reference is set up to source formula/resolvent root and R reference is set up to the examined formula. It doesn't create any copy and after usage it is restored. It is performed self-resolution by OptimizeX procedure until it has a logical value and if the value is false then the formula is not added to the set of resolvents.

6 Computer application.

6.1 General information.

As it has been noticed, the application, which should be able to demonstrate the capabilities of general resolution, was produced. It is called GEneral Resolution Deductive System and it is implemented on 32-bit Windows (95,98,NT) platform. If the system is accompanying the thesis, you will probably find thesis.exe self-extracting file, which can be executed and it creates Thesis directory with Program and Docs directories. The Docs directory contains this paper and Program is consisted of application source codes, executable file of the application, font and examples. The program executable file has the name GERDS.exe. The user interface of GERDS is quite simple. It is a MDI (Multiple document interface) application, which means that the user has possibility to open more than one set of source formulas at once. As it was noticed in the previous section, every independent frame has two parts – Editor for source formulas and Output for results of inference. Here is an example of the frame.



The upper window - editor can accept source formulas of the form defined by the BNF in the section 2. The lower window – output shows the results of the inference. It may vary depending on the user demands. It shows the resolvents and their premise numbers. Let's have a look to detailed description of source set and results. You can create new or open an existing frame by selecting command from the File menu as well as other standard operations (Save, close, exit).

You can use the Window menu to select particular window or reorganize the window order. The Help menu gives you essential information about the program and keyboard layout for special character.

6.2 Input and Output.

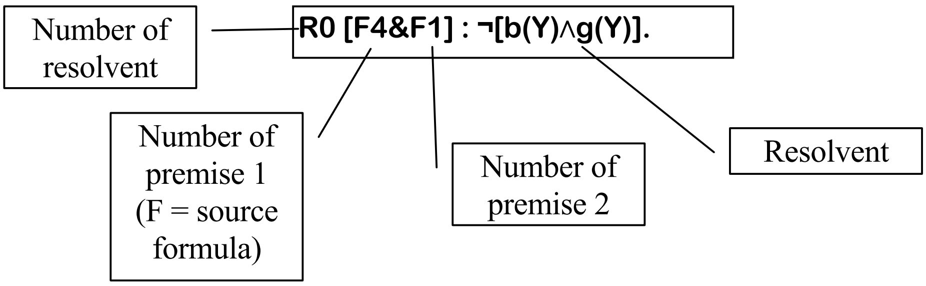
The editor (input) window accepts source formulas and goals by syntax of BNF from the section 2.1. Source formulas end by dot and one or more blank character (space, return or other character with ASCII code between 1 and 32). The goal (query) is introduced by '?-' string and may occur several times in the editor. Every goal is proved using all preceding source formulas. Consider next example.

Example 6.1

a(X)∧g(X)→b(X).
b(X)∧g(X)→c(X).
a(a). ?-g(a).
g(a).
?-c(Y).
There are two goals here.

The result (output) window may contain a list of source formulas and goals (already negated), sequence of resolvents, unsimplified forms of resolvents, list of resolvents and some statistics. The items in the sequence of resolvents are of the form by this example:

Let's see to an example of the result to the previous source set.



Example 6.2

Source formulas (axioms) :
F0 : [a(X)∧g(X)→b(X)].
F1 : [b(X)∧g(X)→c(X)].
F2 : a(a).
F3 (¬query) : ¬g(a).

Source formulas (axioms) :
F0 : [a(X)∧g(X)→b(X)].
F1 : [b(X)∧g(X)→c(X)].
F2 : a(a).
F3 : g(a).
F4 (¬query) : ¬c(Y).

R0 [F4&F1] : ¬[b(Y)∧g(Y)].
R1 [R0&F3] : ¬b(a).
R2 [R1&F0] : ¬[a(a)∧g(a)].
R3 [R2&F3] : ¬a(a).
[R3&F2] : YES.
Y = a.

R4 [R2&F2] : ¬g(a).
[R4&F3] : YES.
Y = a.
Solving time : 0.22 s. Used memory : 52956 B.

The resolvent with YES means the false resolvent and Y = a represents the variable assignment performed during refutation (a = constant). Notice that the first goal has no solution, since g(a) follows after the goal.

In both windows you can choose Edit submenu from the application menu and perform standard operations – select text, copy, cut, clear and paste selection, clear output and select font.

Logical connectives can be added to the editor by holding Ctrl+Alt key and by pressing appropriate key:

Q - forall, W - exists, E - equivalence, I - implication,

C - conjunction, D - disjunction, N - negation, F - not equal,

L - less or equal, G - greater or equal.

6.3 Proof.

There are three additional menus for proving process. The Output menu defines formatting characteristics of the output. It has following items, which can be checked off:

Axioms – adds source set formulas with labels (goals are negated).

Progress – shows the sequence of the proof.

Sources – adds labels of premises.

Resolvents – summarizes generated resolvents.

Time, Memory – adds time consumed and memory currently used by the application.

Unsimplified – shows all resolvents in unsimplified form.

Statistics – shows statistics about the proving process.

The Prove theorem menu contains some specifications to a proof. The stop item causes the break of an inference. The linear search, breadth-first search, modified linear search starts the proving using appropriate strategy. Modified linear search utilizes derivations not only from goal, but also from source set formulas. Last four items are alternatively clickable in the panel below menu. One from further three radio items is selectable. It allows you to choose the proof without resolvent redundancy check, with consequence check and equivalence check. Similar function belongs to next items: without restriction strategy, filtration strategy or support-set strategy.

The Unification menu consists of quantification – if checked off , the quantifiers are significant else they have the same meaning of universality. Further great cut command provides partial resolution and general cut provides general resolution else it is carried out the restricted resolution. Next three items determine, how many resolvents have to be generated from two premises. Exit on first unused creates one resolvents of any type. Exit on first match creates one resolvent, which doesn't degrade to true or false or which is not redundant. Exit on last match performs all possible resolution form two premises. Last two items determine if one refutation or all refutations will be done.

7 Examples.

7.1 Demonstration.

At the end, let's consider some interesting examples generated by GERDS. We start with simple propositional examples. All examples were produced using Pentium 100 machine.

Example 7.1

Source formulas (axioms) :
F0 : a(X). **F1 : b(X).**
F2 (¬query) : ¬[a(X)∧b(X)].

R0 [F2&F1] : ¬a(X).
[R0&F0] : YES.
R1 [F2&F0] : ¬b(X).
[R1&F1] : YES.

Example 7.2

Source formulas (axioms) :
F0 : a∧¬b.
F1 : ¬a∧b.
F2 (¬query) : ¬c.

At first, we use linear strategy, which doesn't lead to a contradiction, because it starts from goal and it is not provable only from combinations of the resolvents derived from goal and source formulas. Nevertheless the set of source formulas is inconsistent, so everything is provable, but we must use modified linear strategy (it uses source set formulas as both premises) or breadth-first search.

R0 [F1&F1] : b.
[F1&F0] : YES.

Of course, the refutation is obtainable by many other derivations.

Example 7.3

[a∧¬b]∨[¬a∧b].
?-¬a∧¬b.
Solution to ?- ¬[¬a∧¬b].
[F1&F1] : ¬[⊤∧¬b]∨¬[⊥∧¬b].
[F1&F0] : ¬[⊤∧¬b]∨ ⊤∧¬b∨ ⊥∧b.
[F1&F0] : ¬[¬a∧ ⊤]∨a∧ ⊥∨¬a∧ ⊤.

Here the solution doesn't exist so we can suppose that the goal is not valid.

Example 7.4

Source formulas (axioms) :
F0 : a∧¬b∧c∧d∨¬a∧b∧¬c∧d.

F1 (¬query) : ¬[¬a∧¬b].

R0 [F1&F0] : b∨¬b∧c∧d.
R1 [F1&F0] : a∨¬a∧¬c∧d.
R2 [F0&F0] : b∧¬c∧d∨¬b∧c∧d.
R3 [F0&F0] : ¬a∧¬c∧d∨a∧c∧d.
R4 [F0&F0] : ¬a∧b∧d∨a∧¬b∧d.
R5 [F0&F0] : a∧¬b∧c∨¬a∧b∧¬c.
It is the next example of a goal, which is not provable.

Example 7.5

Source formulas (axioms) :
F0 : a→b∧g. F1 : b∧g→c. F2 : b∧g→a. F3 : c→b∧g.
F4 (¬query) : ¬[a↔c].

R0 [F4&F3] : a∨b∧g.
R1 [R0&F4] : b∧g∨¬c.
R2 [R1&F2] : ¬c∨[g→a].
R3 [R2&F1] : [g→a]∨¬[b∧g].
R4 [R3&F4] : ¬g∨¬[b∧g]∨¬c.
R5 [R4&F3] : ¬b∨¬c.
R6 [R5&F4] : ¬b∨a.
R7 [R6&F3] : a∨¬c.
R8 [R7&F4] : ¬c.
R9 [R8&F4] : a.
R10 [R9&F0] : b∧g.
R11 [R10&F1] : g→c.
R12 [R11&F4] : ¬g∨¬a.
R13 [R12&F0] : ¬a.
R14 [R13&F4] : c.
[R14&R8] : YES.
Solving time : 0.48 s.

Source formulas (axioms) :
F0 : a↔b∧g. F1 : b∧g↔c. F2 (¬query) : ¬[a↔c].

R0 [F2&F1] : [¬a∨¬[b∧g]]∧[a∨b∧g].
R1 [R0&F2] : ¬[b∧g]∨c.
R2 [R1&F0] : ¬g∨c∨¬a.
R3 [R2&F2] : ¬g∨¬a.
R4 [R3&F2] : ¬g∨c.
R5 [R4&F0] : c∨¬a.
R6 [R5&F2] : ¬a.
R7 [R6&F2] : c.
R8 [R7&F1] : b∧g.
R9 [R8&F1] : g↔c.
R10 [R9&F2] : [g∨a]∧[¬g∨¬a].
R11 [R10&F1] : a∨[b↔c].
R12 [R11&F1] : a∨¬c∨¬g.
R13 [R12&F2] : ¬c∨¬g.
R14 [R13&F2] : ¬g∨a.
R15 [R14&F1] : a∨¬c.
R16 [R15&F2] : ¬c.
R17 [R16&F2] : a.
[R17&R6] : YES.
Solving time : 0.54 s.

Last two inferences were an example of strongly non-clausal resolution. As stated, the proof of **a↔c** can be done both from implicative and equivalence based axioms.

Example 7.6

F0 : [a(X)∧g(X)→b(X)].
F1 : a(a).
F2 : g(c).
F3 (¬query) : ¬b(a).

R0 [F3&F0] : ¬[a(a)∧g(a)].

R1 [R0&F1] : ¬g(a).
Solving time : 0.06 s.
b(a) is not provable, because g(a) doesn't hold. When we add it, the proof exists.

Source formulas (axioms) :

F0 : [a(X)∧g(X)→b(X)].	**F1 : a(a).**
F2 : g(c).	**F3 : g(a).**
F4 (¬query) : ¬b(a).	

R0 [F4&F0] : ¬[a(a)∧g(a)].	**R1 [F3&F0] : [a(a)→b(a)].**
R2 [F2&F0] : [a(c)→b(c)].	**R3 [F1&F0] : [g(a)→b(a)].**
R4 [R3&F4] : ¬g(a).	**R5 [R3&F3] : b(a).**
R6 [R1&F4] : ¬a(a).	**[R6&F1] : YES.**

Solving time : 0.27 s.
There are interesting resolvents with implication, which show the lucidity of the proof (in contrast with clausal resolution), in this breadth-first search proof above.

Example 7.7

Source formulas (axioms) :

F0 : [a(X)∧g(X)→b(X)].	**F1 : [b(X)∧g(X)→c(X)].**
F2 : a(a).	**F3 : g(a).**
F4 (¬query) : ¬c(Y).	

R0 [F4&F1] : ¬[b(Y)∧g(Y)].	**R1 [F3&F1] : [b(a)→c(a)].**
R2 [F3&F0] : [a(a)→b(a)].	**R3 [F2&F0] : [g(a)→b(a)].**
R4 [F1&F0] : [[g(X)→c(X)]∨¬[a(X)∧g(X)]].	
R5 [R4&F4] : [¬g(X)∨¬[a(X)∧g(X)]].	**R6 [R4&F3] : [c(a)∨¬a(a)].**
R7 [R3&F3] : b(a).	**R8 [R3&F1] : [¬g(a)∨c(a)].**
R9 [R3&R0] : ¬g(a).	**R10 [R1&F4] : ¬b(a).**
R11 [R1&R7] : c(a).	
[R11&F4] : YES.	**Y = a.**
[R10&R7] : YES.	**Y = a.**
R12 [R10&R2] : ¬a(a).	
[R9&F3] : YES.	**Y = a.**
[R7&R10] : YES.	**Y = a.**
[R12&F2] : YES.	**Y = a.**

Solving time : 1.72 s.

This example is the first one with a goal requiring variables. There were five possibilities to refute the source set.

Example 7.8

Source formulas (axioms) :

F0 : [a(X)→a(X+1)].	**F1 : a(0).**
F2 (¬query) : ¬a(5).	

R0 [F1&F0] : a(1).	**R1 [R0&F0] : a(2).**
R2 [R1&F0] : a(3).	**R3 [R2&F0] : a(4).**
R4 [R3&F0] : a(5).	**[R4&F2] : YES.**

Solving time : 0.05 s.
This is a sample of the usage of infix operators.

Example 7.9

Source formulas (axioms) :
F0 : 1<2. **F1 : 2<3.**
F2 : 3<4. **F3 : 4<5.**
F4 : 5<6. **F5 : [X<Y∧Y<Z→X<Z].**
F6 (¬query) : ¬1<5.

R0 [F6&F5] : ¬[1<Y∧Y<5]. **R1 [R0&F3] : ¬1<4.**
R2 [R1&F5] : ¬[1<Y∧Y<4]. **R3 [R2&F2] : ¬1<3.**
R4 [R3&F5] : ¬[1<Y∧Y<3]. **R5 [R4&F1] : ¬1<2.**
[R5&F0] : YES.
Solving time : 0.22 s.

This proof illustrates transitivity, which is well solved by the application.

Example 7.10

Source formulas (axioms) :
F0 : [a(X)∧g(Z)→b(X)]. **F1 : [b(X)∧g(Z)→c(X)].**
F2 : a(a). **F3 : g(30).**
F4 (¬query) : ¬[c(Y)∨g(Y)∨b(Y)].

R0 [F4&F4] : ¬[g(Y)∨b(Y)]. **R1 [F4&F4] : ¬[c(Y)∨b(Y)].**
R2 [F4&F4] : ¬[c(Y)∨g(Y)].
[F4&F3] : YES. **Y = 30.**
[F3&F4] : YES. **Y = 30.**
R3 [F3&F0] : [a(X)→b(X)]. **R4 [R3&F4] : ¬a(X).**
R5 [R3&F2] : b(a).
[R2&F3] : YES. **Y = 30.**
[R1&R5] : YES. **Y = a.**
[R0&F3] : YES. **Y = 30.**
[R0&R5] : YES. **Y = a.**
[R5&F4] : YES. **Y = a.**
[R5&R1] : YES. **Y = a.**
[R5&R0] : YES. **Y = a.**
[R4&F2] : YES. **Y = a.**
Solving time : 0.54 s.

Next example produces the Fibonacci sequence.

Example 7.11

Source formulas (axioms) :
F0 : [f(I, A) ∧ f(I+1, B) → f(I+2, A+B)].
F1 : f(0, 1). **F2 : f(1, 1).**
F3 (¬query) : ¬f(15, X).

R0 [F2&F0] : [f(2, B)→f(3, 1+B)]. **R1 [F1&F0] : [f(1, B)→f(2, 1+B)].**
R2 [R1&F2] : f(2, 2). **R3 [R0&R2] : f(3, 3).**
R4 [R3&F0] : [f(4, B)→f(5, 3+B)]. **R5 [R2&F0] : [f(3, B)→f(4, 2+B)].**
R6 [R5&R3] : f(4, 5). **R7 [R4&R6] : f(5, 8).**
R8 [R7&F0] : [f(6, B)→f(7, 8+B)]. **R9 [R6&F0] : [f(5, B)→f(6, 5+B)].**
R10 [R9&R7] : f(6, 13). **R11 [R8&R10] : f(7, 21).**

R12 [R11&F0] : [f(8, B)→f(9, 21+B)].
R13 [R10&F0] : [f(7, B)→f(8, 13+B)].
R14 [R13&R11] : f(8, 34). **R15 [R12&R14] : f(9, 55).**
R16 [R15&F0] : [f(10, B)→f(11, 55+B)].
R17 [R14&F0] : [f(9, B)→f(10, 34+B)].
R18 [R17&R15] : f(10, 89). **R19 [R16&R18] : f(11, 144).**
R20 [R19&F0] : [f(12, B)→f(13, 144+B)].
R21 [R18&F0] : [f(11, B)→f(12, 89+B)].
R22 [R21&R19] : f(12, 233). **R23 [R20&R22] : f(13, 377).**
R24 [R23&F0] : [f(14, B)→f(15, 377+B)].
R25 [R22&F0] : [f(13, B)→f(14, 233+B)].
R26 [R25&R23] : f(14, 610). **R27 [R24&R26] : f(15, 987).**
[R27&F3] : YES. **X = 987.**
Solving time : 1.29 s.

We found the fifteenth Fibonacci number, which is 987 (if zero number is 1).

Another known sequence may be produced using factorial function.

Source formulas (axioms) :

F0 : [f(X, Y)→f(X+1, Y*(X+1))]. **F1 : f(1, 1).**
F2 (¬query) : ¬f(10, Y).

R0 [F1&F0] : f(2, 2). **R1 [R0&F0] : f(3, 6).**
R2 [R1&F0] : f(4, 24). **R3 [R2&F0] : f(5, 120).**
R4 [R3&F0] : f(6, 720). **R5 [R4&F0] : f(7, 5040).**
R6 [R5&F0] : f(8, 40320). **R7 [R6&F0] : f(9, 362880).**
R8 [R7&F0] : f(10, 3628800).
[R8&F2] : YES. **Y = 3628800.**

Now let's have a look into simple quantified examples. As it was noticed, these simple examples are workable by the application.

At first, consider several combinations of the base case, which illustrate the behaviour of existential variables.

Example 7.12

F0: ∀X∃Y p(X,Y).
?-∃Y∀X p(X,Y). ?-∃Y∃X p(X,Y). ?-∀X∃Y p(X,Y). ?-∀Y∀X p(X,Y).
Solution to #1.
Solution to #2.
[F1&F0] : YES.
Solution to #3.
[F1&F0] : YES.
Solution to #4.

Case 1 and 4 have no solution, since they do not imply from F0. The opposite case follows.

∃Y∀X p(X,Y).
?-∀X∃Y p(X,Y). ?-∃X∃Y p(X,Y). ?-∀X∀Y p(X,Y).
Solution to #1.
[F1&F0] : YES.
Solution to #2.
[F1&F0] : YES.
Solution to #3.

Here case 1 and 2 lead to refutation. If we change the axiom, this result is obvious.

∃Y∃X p(X,Y).
?-∀X∃Y p(X,Y). ?-∃X∃Y p(X,Y). ?-∀X∀Y p(X,Y).
Solution to #1.
Solution to #2.
[F1&F0] : YES.
Solution to #3.

These examples show the simplest cases, but now let's have a look into more complicated instance, where it is needed to perform more than one pass to unify two atoms.

Example 7.13

∀Z∃Y p(X,Y,Z). **?- ∀Z∃Y p(X,Y,Z).**
[F1&F0] : YES.

The unifying algorithm strikes on a problem, when it unifies these two atoms. It takes X without problems, but when it tries to substitute universal Y (with respect to polarity!) from F1, the existential Y from F0 has one superior variable Z. Z is not substituted yet, so we temporarily continue with next argument - Z variable. It can be substituted with existential Z (which has no superior variable) and then the next pass follows. In this pass, we are able to substitute Y, since Z is already substituted.

Next samples show instances similar to example 7.12.

∀Z∃Y p(X,Y,Z). **?- ∃Y∀Z p(X,Y,Z).**
Solution to #1.
and
∃Y∀Z p(X,Y,Z). **?- ∀Z∃Y p(X,Y,Z).**
Solution to #1.
[F1&F0] : YES.

Example 7.14

This example is an illustration of more complicated formulas with existential variables.

∀X a(X) → ∃X b(X). ∀X a(X).
?- b(a).
Solution to ¬b(a).

∃X a(X)→∀X b(X). a(a).
?- b(c).
Solution to ¬b(c).
R0 [F2&F0] : ∀X ¬a(X).
[R0&F1] : YES.

7.2 Comparison of GERDS with the clausal resolution through SPASS.

The Max-Planck-Institut für Informatik in Germany has developed effective and really suitable software tool for theorem proving. Since it is a full first-order logic theorem prover with equality and GERDS currently doesn't support equality all examples will be based on non-equality problems. Comparing problems are taken from the SPASS [3] and they are specially selected to demonstrate advantages of non-clausal prover (GERDS). It is clear that SPASS is more practical prover than GERDS, nevertheless the GERDS has also some usage, I hope.

Original notation of SPASS for Pelletier's Problem No. 71:

formula(equiv(P1,equiv(P2,equiv(P3,equiv(P4,equiv(P5,equiv(P1,equiv(P2,equiv(P3,equiv(P4,P5)))))))))).

and GERDS notation:

?-p1↔[p2↔[p3↔[p4↔[p5↔[p1↔[p2↔[p3↔[p4↔p5]]]]]]]]].

This simple sequence of equivalencies seems to be valid, but how to prove it? If we utilise clausal prover (SPASS) 32 clauses are generated from the goal. Then it kept 494 clauses during the proving process. In contrast the GERDS produced a short proof by general resolution (including SR-check of the resolvent):

Solution to ¬[p1↔p2↔p3↔p4↔p5↔p1↔p2↔p3↔p4↔p5].
SR_check:¬[p3↔p4↔p5↔p3↔p4↔p5]∨[p3↔p4↔p5↔¬[p3↔p4↔p5]]∨[p3↔p4↔p5↔¬[p3↔p4↔p5]]∨¬[p3↔p4↔p5↔p3↔p4↔p5].
SR_check:¬[p4↔p5↔p4↔p5]∨[p4↔p5↔¬[p4↔p5]]∨[p4↔p5↔¬[p4↔p5]]∨¬[p4↔p5↔p4↔p5]∨[p4↔p5↔¬[p4↔p5]]∨¬[p4↔p5↔p4↔p5]∨¬[p4↔p5↔p4↔p5]∨[p4↔p5↔¬[p4↔p5]].
SR_check:¬[p5↔p5]∨[p5↔¬p5]∨[p5↔¬p5]∨¬[p5↔p5]∨[p5↔¬p5]∨¬[p5↔p5]∨¬[p5↔p5]∨[p5↔¬p5]∨[p5↔¬p5]∨¬[p5↔p5]∨¬[p5↔p5]∨[p5↔¬p5]∨¬[p5↔p5]∨[p5↔¬p5]∨[p5↔¬p5]∨¬[p5↔p5].
SR_check: ⊥.
[F0&F0] : YES.
Solving time : 0.00 s. Used memory for proof: 920 B.

Let’s have a look to a non-propositional example (Pelletier's Problem No. 68):

It consist of these axioms:

∀U∀V t(i(U,i(V,U))).
∀U∀V∀W t(i(i(U,i(V,W)),i(i(U,V),i(U,W)))).
∀U∀V t(i(i(V,U),i(n(U),n(V)))).
∀U∀V [t(i(U,V))∧[t(U) → t(V)]].

and this goal:

?-∀U t(i(U,n(n(U)))).

The SPASS had a short proof here, it kept 3 only clauses, but GERDS is also efficient:

Solution to ∃U ¬t(i(U, n(n(U)))).
[F4&F4] : ⊥∨ ⊤. – improductive resolvent.
[F4&F3] : ⊥∨[⊥∧[t(U)→t(n(n(U)))]].
[F4&F3] : YES.
Solving time : 0.00 s. Used memory for proof: 2592 B.

It was added unsimplified form of resolvent, in order to make the resolution more understandable.

And another non-propositional example (Pelletier's Problem No. 60):

?-∀U [f(U,f(U)) ↔ ∃V[∀W [f(W,V)→ f(W,f(U))]∧f(U,V)]].
Solution to ∃U¬[[f(U, f(U))→∃V[∀W[f(W, V)→f(W, f(U))]∧f(U, V)]]∧[∃V[∀W[f(W, V)→f(W, f(U))]∧f(U, V)]→f(U, f(U))]].
[F0&F0] : ∃U¬[[⊥→[∀W[f(W, f(U))→f(W, f(U))]∧ ⊥]]∧[∃V[[f(U, V)→ ⊥]∧f(U, V)]→ ⊥]]∨∃U¬[[⊤→[∀W[f(W, f(U))→f(W, f(U))]∧ ⊤]]∧[∃V[[f(U, V)→ ⊤]∧f(U, V)]→ ⊤]].
R0 [F0&F0] : ∃U, W¬[f(W, f(U))→f(W, f(U))].
[R0&R0] : ∃U, W¬[⊥→ ⊥]∨∃U, W¬[⊤→ ⊤].
[R0&R0] : YES.
Solving time : 0.00 s. Used memory for proof: 2308 B.

SPASS has kept 8 clauses in this example and GERDS has produced one resolvent.

Pelletier's Problem No. 50 – SPASS kept 4 clauses:

?-∀U [f(a,U) ∨ ∀V f(U,V)] → ∃W∀X f(W,X).
Solution to ¬[∀U[f(a, U)∨∀V f(U, V)]→∃W∀X f(W, X)].
R0 [F0&F0] : ∀W#1∃X#1 ¬f(W#1, X#1).
[R0&F0] : YES.
R1 [F0&R0] : ∀X¬[∀V f(X, V)→∃W∀X f(W, X)].
[R1&R0] : YES.
Solving time : 0.00 s. Used memory for proof: 1648 B.

Pelletier's Problem No. 14 (from SPASS):

SPASS notation goal: formula(equiv(equiv(P,Q),and(or(Q,not(P)),or(not(Q),P)))).

?-[p ↔ q]↔[[q∨¬p]∧[¬q∨p]].
Solution to ¬[p↔q↔[q∨¬p]∧[¬q∨p]].
[F0&F0] : ¬[⊤↔q↔[q∨ ⊥]∧[¬q∨ ⊤]]∨¬[⊥↔q↔[q∨ ⊤]∧[¬q∨ ⊥]].
Before:¬[⊤↔ ⊤]∨¬[⊥↔ ⊥]∨¬[⊥↔ ⊥]∨¬[⊤↔ ⊤].
SR-check: ⊥.
[F0&F0] : YES.
Solving time : 0.00 s. Used memory for proof: 592 B.

8 Conclusions.

The last chapter analyzed some examples, which were the best approach for reader to understand the theoretical power of general resolution. Although the general resolution is not efficient in comparison with clausal and Horn logic, it preserves the sequence of the proof. Achieved solving times are controversial. It is obvious, that these times grow quicker than for clauses. Even if we use some techniques of avoiding redundancy, the efficiency is still low for the serious usage in knowledge representation. Nevertheless, the proposed theoretical system may be a good source of training for persons interested in deductive systems as well as in logic generally.

The existence handling methods are partially satisfied in the application, which is one of the results of the thesis, and they were shown in examples as functional extensions of ground general resolution. Also the check of consequence by self-resolution, which was proposed by the thesis, is strong result making the proving by general resolution applicable to machine-performed deduction and not only applicable for intuitive human-performed proofs.

There are many further problems in theorem-proving and resolution techniques as its main branch. That's why the thesis may lead to continual research in theory of resolution as well as in progressive sectors of logic (Fuzzy logic, Objective-oriented logics). Though it will not continue directly, it is an excellent starting point for designing another knowledge-based systems, which, I believe, will be the objective of my future work. It lies mainly in programming the application. It wasn't an easy job to design and realize the GERDS and unfortunately, it is not so visible result as I imagined. The important contribution of the thesis is to remind, that there are essential items in logic, that are not emphasized in conventional studies, and it is interesting to deal with them.

9 References.

[Ba97] Bachmair L., Ganzinger H.: **A theory of resolution.** Technical report of the Max-Planck-Institut für Informatik, 1997.

[Be71] Berztiss A.T.: **Data structures.** Academic press, 1971.

[Ce81] Čechák V.: **Co víte o moderní logice.** Horizont, 1981.

[Dv92] Dvořák S.: **Dekompozice a rekurzivní algoritmy.** Grada, 1992.

[Ji88] Jinoch J., a kol.: **Programování v jazyku Pascal.** SNTL, 1988.

[Kl67] Kleene S.C. **Mathematical logic.** John Willey & Sons, 1967.

[Lu95] Lukasová A.: **Logické základy umělé inteligence.** Učební texty Ostravské Univerzity, 1995,1997.

[Ma93] Mařík V., Štěpánková O., Lažanský J.: **Umělá Inteligence (1), (2).** Academia 1993,1997

[No99] Novák, V., Perfilieva I., Močkoř J.: **Mathematical principles of fuzzy logic.** Kluwer Academic Publishers, 1999.

[RaN1] Ramsay A.: **Formal methods in artificial inteligence.** Cambridge university press.

[Ri89] Richards T.: **Clausal Form Logic.** Addison-Wesley,1989.