Specifying Sequent Calculi Rules for Managing Some Redundancies in Proof Search

Lutovac A. Tatjana

Abstract – A central aspect of proof search is the identification and control over various forms of redundancies in the search space. We investigate systematic techniques for managing some redundancies in proof search in sequent calculi. This paper is a summary of some results of our investigation. In particular we have enriched inference rules with some additional information about status the search in order to preclude some redundant or useless choices which would otherwise be allowed in the standard sequent system. We have developed a method for detection of redundant and eliminable formulae from a given sequent proof and an algorithm for ensuring termination ie. for eliminating (infinite) loops during a backward sequent calculi proof search.

Key words: affine logic, backward proof search, linear logic, loops, redundant formulae, sequent calculus

1. *Introduction*

It is well known that logic programming may be thought of as the application of the techniques of mathematical logic to programming tasks. Logic programs may be considered as collections of formulas and their computation may be identified as *searching for proofs*: given a program $\mathcal{P}$ and a goal G we attempt to satisfy G by *searching for a proof* of $\mathcal{P} \rightarrow G$ using the inference rules of a given logic.

Proof search is the name given to the study of the construction, if possible, of proofs of a given logical assertion. In systems known as sequent calculi logical assertions are presented in the form of sequents.

A *sequent* is a pair denoted by $\Gamma \vdash \Delta$, where both the *antecedent* Γ and the *succedent* Δ are sequences of formulae. The intuitive meaning of $\Gamma \vdash \Delta$ is: if all of the formulae in Γ are true, then at least one formula in Δ is true.

Manuscript received December, 11, 2006.

Author is with Department of Applied Mathematics, Faculty of Electrical Engineering, University of Belgrade, Belgrade, Serbia (e-mail: tlutovac@eunet.yu).

A sequent calculus system gives a set of rules for manipulating sequents, in the form of set of sequent rules. Sequent calculus rules generally are written as follows.

$$\frac{sequent_1,\ sequent_2,\ \ldots sequent_k}{sequent}\ \text{rule-name} \qquad k = 0, 1, ...$$

The sequents above the line in a rule are called *premises* of the rule and the sequent below the line is called the *conclusion*.

A *proof* of a sequent $\Gamma \vdash \Delta$ is a tree whose nodes are labelled with sequents such that the root node is labelled with $\Gamma \vdash \Delta$ (which is then called the *end-sequent*), the internal nodes are instances of one of the inference rules, and the leaf nodes are labelled with axioms.

A sequent $\Gamma \vdash \Delta$ is *provable* in the sequent calculus formalization of a logic (or logical fragment) if there is a proof with $\Gamma \vdash \Delta$ as the end- sequent.

Bottom-up ie. *backwards* proof search consists of building the proof from the given end-sequent. The end-sequent is reduced by reversed (conclusion-to-premise) application of a sequent calculus rule to produce subgoal branches.

The theory of the exploration of the search space generated by inference rules used in this way is subject of proof search . A number of computational difficulties arise in designing algorithms for proof search . Prominent among these is the need, in order to minimize complexity, to control and/or eliminate as much redundancy from the search space as possible.

There have been a variety of proof-theoretic techniques used to analyze and design strategies for efficient sequent calculi proof search in theorem proving and logic programming [13, 16, 15]. It is notable that many of the existing strategies are all rather sophisticated and involve complex manipulations of proofs. Many are restricted to particular logic or classes of formulae. Almost all are designed for analysis on paper by a human and many of them are ripe for automation, being formally defined in

precise detail, and yet somewhat overwhelming for humans.

We investigate systematic and automated-oriented techniques for managing some forms of redundancies in proof search in sequent calculi. This paper is a summary of some results, being developed, presented and formally proved in [7][1], on the development of systematic techniques:
- for detection of redundant and eliminable formulae from a given sequent proof and
- for ensuring termination ie. eliminating infinite loops during a backward proof search.

Our solutions are based on the widely used practice of including in the representation of the sequents some additional information about the status of the search.

Our work on detection of redundant parts of sequent proof is motivated by the fact that none of the existing algorithms for efficient implementation of proof search (in linear logic, at least) can distinguish redundant formulae that can be freely, unconditionally eliminated from the proof from those redundant formulae whose elimination lead to an invalid proof, as far as we are aware. Our mechanism makes such a distinction. Furthermore it allows selection in a sense that a redundant (sub)formula can be either eliminated or replaced by an arbitrary formula. This induces a class of equivalent formulae (in terms of provability) and a class of equivalent proofs modulo the redundant parts. This allows, for example, more flexible reuse of previously successful searches and is potentially useful for implementations.

There have been a variety of systems for preventing and detecting infinite loops during proof search. Surprisingly, no loop detection mechanism has been developed or described in the literature (as far as we are aware) for resource sensitive logic (such as, for example, linear and affine logic). We have developed a (terminating) sequent calculi with loop-detection mechanism for intuitionistic, propositional[2] affine logic. We have followed the overall approach of [3] and [4], i.e., to incorporate the loop-detection mechanism into the sequent rules. The proposed conditions are independent of the search strategy used and no explicit loop checking is needed in the interpreter.

[1] Parts of this material are presented in [8, 9].

[2] Quantifier free.

Given the evolution of programming languages towards higher and higher level languages, with a corresponding increase in the computational power of the execution models of these languages, a natural demand is a wide range of expressive logics and more powerful inference facilities in which to write programs. That is why our starting point is propositional linear logic and affine logic. Linear logic is a refinement of classical logic, in that there is a fragment of linear logic which has precisely the same properties as classical logic; at the same time however, linear logic contains features which are not present in classical logic. In essence, these features are due to removing the rules for contraction and weakening and reintroducing them in a controlled manner. It is simpler and more natural to express certain resource management problems in linear logic than in classical logic. For example, the property of having two dollars we may represent by the LL conjunction $\$1 \otimes \$1$. In classical logic, this would be represented by $\$1 \wedge \$1$, which is equivalent to $\$1$, ie. that having two dollars is equivalent to one dollar, which is clearly nonesential. However, in linear logic $\$1 \otimes \$1$ and $\$1$ are not equivalent, which is more appropriate. For this reason linear logic is often described as a logic of resources rather than logic of truth (such as classical logic) in that different amounts of the same thing are considered to be different.

We are interested in affine logic because of its relationship to logic programming. Due to the presence of the weakening rules[3], there are many problems where affine logic is better suited than linear logic. Propositional affine logic [5] is decidable[4] , and so it seems reasonable to expect a complete proof search procedure with a loop detection mechanism.

This paper is organized as follows. In Section 2 we briefly explain our mechanism for detection and elimination of redundant formulae in sequent proof.

[3] While proofs in linear logic must use each linear formula exactly once, proof derivations in affine logic use each affine formula at most once.

[4] Let us recall that a logical system is *decidable* iff there exists an algorithm such that for every well-formed formula in that system there exists a maximum finite number N of steps such that the algorithm is capable of deciding in less than or equal to N algorithmic steps whether the formula is (semantically) valid or not valid.

In Section 3 we illustrate our solution for prevention of infinite loops in propositional affine logic. In Section 4 we present our conclusions.

2. Redundant formulae in sequent proofs

Proof search often involves managing information which later, when the proof is completed, turns out to be redundant. For example, the sequent $p, q \vdash p, r, s$ is provable in classical (and affine) logic. However q, r and s are redundant, and the "core" provable sequent is $p \vdash p$. More complex examples involve choosing between subformulae.

Example 1. Consider the (linear logic) proof Π below:

$$\Pi: \quad \dfrac{\dfrac{}{t \vdash t}\,Ax \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{r \vdash r}\,Ax}{r \vdash ?p, r}\,?wR}{r \vdash ?q, ?p, r}\,?wR}{r \vdash ?q \wp ?p, r}\,\wp R}{r \vdash (?q \wp ?p) \oplus s, r}\,\oplus R}{r, t \vdash t \otimes ((?q \wp ?p) \oplus s), r}\,\otimes R \qquad \dfrac{?}{r, t \vdash t, r}$$

What is the core of this proof ie. what is the minimal set of formulae which guaranties success of the proof? The (sub)formulae p, q and s are *unused*, but only s can be freely deleted from the proof while formulae p and q cannot be simultaneously eliminated. Elimination of the whole formula $(?q \wp ?p) \oplus s$ will disable proof branching ie. distribution of formulae across the multiplicative branches of the proof. Elimination of the subformula $?q \wp ?p$ will also lead to the unprovable sequent (on the right-hand side above). So, we have that p, q and s are unused and that p and q cannot be simultaneously eliminated from the proof. For each unused atom we have three possibilities: to omit it from the proof; to leave the atom unchanged or to replace it with an arbitrary formula. So proof Π can be thought of as a template for $(3^2-1)\cdot 3$ proofs (ie. some variations of the given proof) which can be generated by alterations of p, q and s. All that proofs do not alter the search strategy used, in that the order of application of the rules is not changed.

This knowledge allows later computations to make use of earlier work. So a proof search strategy can retain the results of a previous successful search and to apply and combine them to a new situation. The knowledge about redundant and eliminable formulae can be potentially useful when composing programs (and hence proofs), for debugging, and for teaching purposes.

2.1 *Our technique for detection of redundant formulae*

In Chapter 4 of [7] we have proposed a mechanism for distinguishing between the necessary and unnecessary formulae in a linear logic proof. We have enriched the standard sequent structure with labels and Boolean constraints as follows:

$\phi_{1,[v_1]}, \phi_{2,[v_2]}, \ldots \phi_{n,[v_n]} \vdash \psi_{1,[w_1]}, \psi_{2,[w_2]}, \ldots \psi_{m,[w_m]} - \mathcal{C}$

where $\mathcal{C}$ is a set of constraints being generated so far on the branch of a proof tree by application of the particular rules and labels $[v_1], \ldots, [w_1], \ldots [w_m]$ trace formulae duplicated by the contraction rules. Labels and constraints allow us to store the necessary information about the usage of formulae in a proof.

We have defined a labelled sequent calculus, called LL^{PRE}, for elimination of redundant formulae in propositional linear logic. LL^{PRE} sequent calculi is defined as shown in Figure 4.1, Chapter 4 of [7]. Examples of some LL^{PRE} rules and the corresponding constraints are given below.

$$\dfrac{- \mathcal{C} \cup \{p > 0\}}{p \vdash p \quad - \mathcal{C}}\,Ax \qquad \dfrac{\Gamma \vdash \psi_{[w]}, \Delta \quad - \mathcal{C} \cup \{\phi_{[w]} \leq \psi_{[w]}\}}{\Gamma \vdash (\phi \oplus \psi)_{[w]}, \Delta \quad - \mathcal{C}}\,\oplus \mathrm{R}$$

$$\dfrac{\Gamma \vdash \phi_{[w]}, \Delta \; - \mathcal{C} \cup \{\phi_{[w]} > 0\} \quad \Gamma' \vdash \psi_{[w]}, \Delta' - \mathcal{C} \cup \{\psi_{[w]} > 0\}}{\Gamma, \Gamma' \vdash (\phi \otimes \psi)_{[w]}, \Delta, \Delta' - \mathcal{C}}\,\otimes R$$

We begin with the empty labels (denoted as $[\,]$) on each formula of the end-sequent and with the empty set $\mathcal{C}$. The labels and constraints generated and accumulated during a proof construction (in the labelled system LL^{PRE}) place some restrictions on the elimination of the corresponding formulae. For example, the intuition underlying the constraint $\phi_{[w]} \leq \psi_{[w]}$ is that elimination of formula $\psi_{[w]}$ is a necessary condition for elimination of formula $\phi_{[w]}$. The intuition underlying the constraint $\phi_{[w]} > 0$ is that formula $\phi_{[w]}$ cannot be entirely eliminated.

A labelled proof tree (generated by the LL^{PRE} sequent rules) is forwarded to the main algorithm, called algorithm PRE, being defined as follows.

Algorithm PRE (input: labelled proof π)

1. *Generate Boolean expressions and constraints on Boolean expressions;*
2. *Calculate possible assignments for Boolean variables;*
3. *If there is an assignment with at least one Boolean variable being assigned the value 0 then:*
   Delete atoms being assigned 0 ie. delete formulae made up of such atoms and the corresponding inferences

   Else EXIT: 'Simplification of proof π is not possible'

Algorithm PRE has to interpret the set of accumulated constraints via the set of Boolean constraints and to find an assignment for Boolean variables. Intuitively, we associate each atom in a proof π with a Boolean variable. Atoms associated with Boolean variables annotated to 0 and the (sub)formulae made up of such atoms can be safely eliminated (ie. deleted) from the proof π.

For example for the proof Π from Example 1, we will have the labelling as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Pi_1}{r_{[]}, t_{[]} \vdash (t_1 \otimes ((?q \wp ?p) \oplus s))_{[]}, r_{1,[]} \quad - \emptyset}\otimes R}{r \vdash (?q \wp ?p) \oplus s, r_1 \quad - \{(?q \wp ?p) \oplus s > 0\}}\oplus R}{r \vdash ?q \wp ?p, r_1 - \{(?q \wp ?p) \oplus s > 0,\ s \leq ?q \wp ?p\}}\wp R}{r \vdash ?q, ?p, r_1 \ - \{(?q \wp ?p) \oplus s > 0,\ s \leq ?q \wp ?p\}}w?R}{r \vdash ?p, r_1 \ - \{(?q \wp ?p) \oplus s > 0,\ s \leq ?q \wp ?p\}}w?R}{r \vdash r_1 \ - \{(?q \wp ?p) \oplus s > 0,\ s \leq ?q \wp ?p\}}w?R}{- \{(?q \wp ?p) \oplus s > 0,\ s \leq ?q \wp ?p,\ r = 1,\ r_1 = 1\}}Ax$$

where the subproof Π_1 is as follows. $\dfrac{- \{t = 1,\ t_1 = 1, t_1 > 0\}}{t \vdash t_1 \ - \{t_1 > 0\}}\ Ax$.

For the proof Π Algorithm *PRE* will extract the following Boolean constraints: $q + p + s > 0$, $s \leq q + p$, $r_1 = 1$, $r = 1$, $t_1 = 1$, $t = 1$. Possible solutions for the (unassigned) Boolean variables are: $(p, q, s) \in \{ (0,1,0), (1,0,0), (1,1,0), (1,1,0), (1,0,1) \}$. Hence, there are five possible simplifications of proof Π. Below we illustrate one of them:

$$(p,q,s) = (0,1,0) \mapsto \dfrac{\dfrac{}{t \vdash t_1}\ Ax \quad \dfrac{\dfrac{}{r \vdash r_1}\ Ax}{r \vdash ?q, r_1}\ ?wR}{r, t \vdash t_1 \otimes ?q, r_1}\otimes R$$

As we already pointed out none of the existing algorithms for efficient implementation of proof search (in linear logic, at least) is able to make the distinction between unused formulae that can be freely eliminated from the proof from those unused formulae whose elimination will cause a 'crash'. Our labelled system makes such a distinction. Furthermore it allows selection in a sense that a redundant (sub)formula can be either eliminated or replaced by an arbitrary formula. Thus, we can get a class of equivalent formulae (in terms of provability) and a class of equivalent proofs modulo the redundant parts. This allows, for example, more flexible reuse of previously successful searches and is potentially useful for implementations.

Our intention was not to find all different proofs of a given sequent but to generate all the concrete simplifications which are instances of a generated proof. Our solution for detection of redundant, eliminable formulae implies elimination which is independent of the search strategy used; elimination which does not alter the search strategy applied and does not require additional proof search ie. redundant formulae remaining in the resulting proof cannot be eliminated without additional proof search. Soundness and completeness of our solution are proved formally in Chapter 4 of [7].

3. Detection and prevention of infinite loops during proof search

It is well known that for many logics, backward proof search in the usual sequent calculi generally does not terminate in general, in the sense of Dyckhoff[1]: 'By "terminating" we mean just that every sequence of steps, in a backward proof search, is finite.' This makes loop detection, where possible, a critical aspect of systems based on backward proof search. For example, consider the (unprovable) linear sequents $!(p \multimap q), !(q \multimap p) \vdash q$ and $!((r \multimap p) \multimap p), r \vdash p$, and the following attempts for proof construction:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\mathcal{P} \vdash q} \quad \dfrac{}{p \vdash p}\ Ax}{\mathcal{P}, q \multimap p \vdash p}\multimap L}{\mathcal{P}, !(q \multimap p) \vdash p}\ !L}{\underbrace{!(p \multimap q), !(q \multimap p)}_{\mathcal{P}} \vdash q}\ !C \quad \dfrac{}{q \vdash q}\ Ax}{\mathcal{P}, p \multimap q \vdash q}\multimap L}{\mathcal{P}, !(p \multimap q) \vdash q}\ !L}{!(p \multimap q), !(q \multimap p) \vdash q}\ !C$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\mathcal{P}_1, r, r \vdash p}}{\mathcal{P}_1, r \vdash r \multimap p}\multimap R \quad \dfrac{}{p \vdash p}\ Ax}{\mathcal{P}_1, r, (r \multimap p) \multimap p \vdash p}\multimap L}{\underbrace{!((r \multimap p) \multimap p)}_{\mathcal{P}_1}, r \vdash p}\ !C, !L$$

The sequent $\mathcal{P} \vdash q$ may continue to occur in the left-hand proof construction. The (sub)formula r may continue to occur in the antecedents during the right-hand proof construction. Note that this is due to the deterministic nature of the proof search process; as in many logic programming languages, the selection of a formula from the antecedent is determined largely by the formula position, and so an interpreter will always handle variants of a given antecedent and a given succedent in exactly the same way (up to renaming variables). Thus far, there is no satisfactory solution to this problem.

There have been a variety of systems [3], [4], [1] for preventing and detecting loops during proof construction. Due to the lack of non-decreasing number of formulae in antecedents, the existing techniques are not directly applicable for resource sensitive logic (such as, for example, linear and affine logic). No loop detection mechanism (apart from a naive history mechanism) has been developed for resource sensitive logic. Naive history

mechanism implies unintelligent searching through the history ie. through is the list of all sequents that have appeared so far on the branch of a search tree. At every step of a proof construction the latest generated sequent is checked to see whether it is a member of the history list. If so, a loop has been generated and so the search backtracks. If not, the history is extended with the sequent and the proof search continues. Implementation of this scheme is clearly inefficient as it requires a great deal of information to be stored. It is important to note that situation illustrated by the right-hand derivation above cannot be detected and solved by the naive history mechanism.

We have identified two reasons for non-termination in propositional linear and affine logic: *simple loops* (ie. appearance of identical sequents in the same branch of a proof tree, as illustrated by the left-hand derivation above) and *special loops* (ie. an infinite repetition of a particular formula, as illustrated by the right-hand derivation above). Our strategy for detecting and preventing loops is twofold. First, we add a simple history list to sequents to allow detection of simple loops, and then introduce machinery that, essentially, turns infinite special loops into simple loops. The idea behind the treatment of special loops is not to make the rule (or sequence of rules) 'responsible' for the special loop inapplicable. The idea is to identify a special loop, and to continue and complete the proof search bearing in mind that some formulae (originated from that special loop) can be used as many times as needed (such as, for example, the formula r in the right-hand example above).

We have proposed sequent calculi rules with zones and side conditions (denoted PIA_{ff}^{Hist-}) for propositional affine logic, as shown in Section 5.7 of [7]. Example of a PIA_{ff}^{Hist-} rule is given below.

$$\frac{\Gamma\,;\,\Upsilon,\,\mathcal{U};\,\Delta-\mathcal{U}\vdash\chi_1\ \blacktriangleright\ f\,;\,\mathcal{H}_1\,;\,h_1}{\Gamma\,;\,\Upsilon;\,\Delta\vdash\chi\ \blacktriangleright\ f\,;\,\mathcal{H}\,;\,h}\ \oplus R\ \Longleftrightarrow\ \mathbf{TEST}(\,\Upsilon;f;\mathcal{H};h;\Delta;\chi_1;\mathcal{U};\mathcal{H}_1;h_1)$$

We have divided the antecedent into zones to isolate and control the non-decreasing part of antecedent ie. to assemble all formulae originated from special loops into a new zone (called $*$context zone) of antecedents. This allows to cut down the amount of storage and checking in the history list. We have extended the sequent rules with the set of side conditions (denoted as procedure TEST). Procedure TEST has to maintain history list, and detect and control simple and special loops. If the latest generated sequent is a member of the history list, procedure TEST make the rule inapplicable (ie. forces the system to backtrack). In the case of special loop, procedure TEST classify all formulae originated from recognized special loop into $*$context zone. As search proceeds, the unrestricted resources of every identified special loop will be classified into the $*context$. Thus, example of the special loop, shown on the right-hand side at the beginning of this section, would be interpreted in the PIA_{ff}^{Hist-} calculi as shown on the left side below.

$$\dfrac{\dfrac{\overbrace{\mathcal{P}_1\,;\,r\,;\quad\ ;\,r,r}\vdash p}{\mathcal{P}_1\,;\quad\ ;\,r\vdash r\multimap p}\multimap R\qquad\dfrac{}{\mathcal{P}_1\,;\quad\ ;\,p\vdash p}\,Ax}{\dfrac{\mathcal{P}_1\,;\quad\ ;\,r,(r\multimap p)\multimap p\ \vdash\ p}{\mathcal{P}_1\,;\quad\ ;\,r\ \vdash\ p}\,!C,!L}\multimap L \qquad \begin{array}{l}\mathcal{P}_1\,;\,r\,;\quad\vdash p\\ \vdots\\ \mathcal{P}_1\,;\,r\,;\quad\vdash p\\ \vdots\\ \mathcal{P}_1\ ;\ ;\ r\vdash p\end{array}$$

$\underbrace{\qquad}_{*\,context\ zone}$

Formula r (being classified in the $*$context) can be further used as many times as needed. An attempt to reiterate the above derivation will cause detection of simple loop (as shown on the right side above). Detection of a simple loop makes the last applied inference rule inapplicable ie. forces the system to backtrack to the most recent decision point and to try to find alternative solution(s). We get failure ie. the sequent at the root of the proof tree is not provable if at any point no rule instance can be applied.

Soundness and completeness of the PIA_{ff}^{Hist-} sequent calculi system as well as the fact that backward proof search in this calculus will always terminate, are proved formally in Chapter 5 of [7].

Providing a terminating procedure for a propositional affine logic could be very useful in the design of tabling mechanisms for linear logic. Another natural extension of this work is to apply the ideas behind the PIA_{ff}^{Hist-} sequent calculi to (fragments of) linear logic. The well-known problem concerning an implementation of logic programming language Forum [11] is connected with the special loops originating from $\perp-$headed implications. Thus far, there is no satisfactory solution this problem. It is our contention that the ideas behind the PIA_{ff}^{Hist-} loop detection mechanism can be used to establish control over the '$\perp-$headed special loops' in Forum.

4. Conclusion

Managing redundancies in sequent calculus proof search is nontrivial. We have briefly presented some mechanisms, being formally defined, developed and proved in [7], for identification and control over different forms of redundancies in sequent calculi proof search . The proposed solutions may contribute to automation of proof search by, for example, fine tuning the search to find one "good" representative of a class of proofs (such as, for example, a class of equivalent proofs modulo redundant formulae).

Our technique for elimination of redundant formulae is limited to sequent proofs and thereby differs from dead-code elimination in functional languages. Developing more general techniques for program slicing and dead-code elimination in advanced logic programming languages are items of future work.

We have developed first terminating sequent system for (a fragment) of propositional affine logic. The perspective to apply the ideas for managing loops in the logic programming language Forum also emphasizes the interest of the results. A natural extension of this work is to apply the ideas behind the PIA_{ff}^{Hist-} sequent calculi to various (fragments of) other resource-sensitive logics.

Our work is intended as a contribution to a library of automatic support tools for managing redundancies in sequent calculi proof search. The proposed strategies and techniques can be implemented and utilized by means of an automated proof assistant such as Twelf [14], possibly in conjunction with constraint logic programming techniques [10].

References

1. Dyckhoff R, " Contraction-free Sequent Calculi for Intuitionistic Logic", The Journal of Symbolic Logic, Vol. 57, No. 3, 1992, pp. 795-807.
2. Harland J, Pym D., " Resource-distribution via Boolean constraints", ACM Transactions on Computational Logic 4:1, 2003, pp. 56-90.
3. Heuerding A., Seyfried M., Zimmermann H., " Efficient Loop-Check for Backward Proof Search in Some Non-classical Propositional Logics", in P. Miglioli at al. (eds.), Proceedings of the 5th International Workshop on Tableaux, Italy, 1996, pp. 210-225.
4. Howe J.M., " Proof Search Issues in Some non-Classical Logics", PhD thesis, School of Mathematical and Computational Sciences, University of St Andrews, 1998.
5. Kopylov A., " Decidability of Linear Affine Logic", Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science 496-504, San Diego, 1995.
6. Lutovac, T., Harland J., " Issues in the Analysis of proof search Strategies in Sequential Presentations of Logics", IJCAR'04 Workshop on Strategies in Automated Deduction, Electronic Notes in Theoretical Computer Science, 125(2), 2005, pp. 115–147.
7. Lutovac T., " Issues in Managing Redundancies in Proof Search", Phd Thesis, School of Computer Science and Information Technology, Science, Engineering and Technology Portfolio, RMIT University, Australia, 2005.
8. Lutovac T., Harland J., " A Redundancy Analysis of Sequent Proofs", International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2005),Germany 2005, $LNAI$ 3702, Springer-Verlag, 2005, pp. 76-90.
9. Lutovac T., Harland J., " Detecting Loops During Proof Search in Propositional Affine Logic", Journal of Logic and Computation, Volume 16, Number 1, 2006, pp. 61-133.
10. Marriot K., Stuckey P., " Programming with Constraints", MIT Press, 1998.
11. Miller D., " Forum: A multiple-conclusion specification-logic", Theoretical Computer Science 165(1), 1996, pp. 201-232.
12. Polakov J., " Linearity Constraints as Bounded Intervals in Linear Logic Programming", in D, Galmiche,(eds.), LICS'04 Workshop on Logic for Resources, Process and Programs (LRPP), 2004, pp. 173-182.
13. Pym D., Harland J., " A Uniform Proof-theoretic Investigation of Linear Logic Programming", Journal of Logic and Computation 4:2, 1994, pp. 175-207.
14. Schürmann C., " Automating the Meta-Theory of Deductive Systems", PhD thesis, Carnegie-Mellon University, 2000.
15. Tammet T., " Proof Search Strategies in Linear Logic", Journal of Automated Reasoning 12, 1994, pp. 273-304.
16. Wallen L., " Automated Proof Search in Non-classical Logic", MIT Press, 1990.