Loop Elimination, a Sound Optimisation Technique for PTTP Related Theorem Proving

Zsolt Zombori

Department of Comptuer Science and Information Theory
Budapest University of Technology and Econcomics
Budapest, Magyar tudósok körútja 2.
H-1117, Hungary
zombori@cs.bme.hu

Péter Szeredi

Department of Comptuer Science and Information Theory
Budapest University of Technology and Econcomics
Budapest, Magyar tudósok körútja 2.
H-1117, Hungary
szeredi@cs.bme.hu

Gergely Lukácsy

Cisco Systems, Inc
Galway, Ireland
glukacsy@cisco.com

Abstract: In this paper we present *loop elimination*, an important optimisation technique for first-order theorem proving based on Prolog technology, such as PTTP and the DLog reasoning mechanism. Although several loop checking techniques exist for logic programs, to the best of our knowledge, we are the first to examine the interaction of loop checking with ancestor resolution. Our main contribution is a rigorous proof of the soundness of loop elimination.

Keywords: resolution, theorem proving, Prolog, PTTP, loop elimination

1 Introduction

Resolution [7] has long been one of the major approaches to automated theorem proving. Besides its theoretical importance, many academic as well as industrial implementations have been built using resolution. Prolog [6] is a programming language that too implements a resolution based inference mechanism. Prolog is highly optimised and has a very high inference rate, thanks to which more complex reasoning systems, such as the Prolog Technology Theorem Prover (PTTP) [8] and the DLog system [5] have been built on top of Prolog. These systems exploit the backtracking mechanism of Prolog to search for a proof of the initial goal. Efficiency is crucial since these systems typically need to explore a huge search space. In this paper we present an optimisation technique called *loop elimination* for Prolog based reasoning. This technique prevents logic programs from trying to prove the same goal over and over again, thus avoiding certain types of infinite loops. Although both of the aforementioned systems already

make use of loop elimination, to the best of our knowledge, there has not yet been any proof of its soundness. Detecting loops to prune the search space for logic programs is not new, see for example [2]. However, the systems that we are interested in extend standard Prolog execution with a technique called *ancestor resolution*, that corresponds to the positive factoring inference rule. In the presence of ancestor resolution, the considerations that trivially justify loop elimination do not hold. Fortunately, as we shall show using a more involved proof, loop elimination is still applicable.

In Section 2 we provide an overview of resolution reasoning and Prolog programming, that will be necessary for understanding the rest of the paper. Section 3 contains our main contribution: we define loop elimination and prove its soundness. We end the paper with some concluding remarks in Section 4.

2 Background

In this section we provide some background information about first-order resolution (Subsection 2.1) and its connection to the Prolog programming language (Subsection 2.2). In Subsection 2.3 we present the Prolog Technology Theorem Prover (PTTP), a complete first-order reasoner built-on top of Prolog. Finally, in Subsection 2.4, we give an overview of DLog, a Description Logic reasoner that implements a PTTP like approach. Due to lack of space, we can only give very sketchy presentation, and we expect the reader to be familiar with the basics of First-Order Logic.

2.1 Resolution Theorem Proving

Resolution [7] is a powerful method for proving first-order theorems. Directly, it is used to check the consistency of a set of first-order clauses, however, all common reasoning tasks – such as entailment analysis – can be easily reduced to consistency check. *Clauses* are first-order formulae satisfying the following properties: all variables are universally quantified, all quantifiers are at the beginning of the formula and the quantifier-free part is a disjunction of *literals*, i.e., possibly negated atomic predicates. It is well known that any set of first-order formulae can be translated into an equisatisfiable set of clauses (for example, see [3]). Since all variables are universally quantified, it is customary to omit the quantifiers. We will do so in the following. Resolution defines two inference rules, called *Binary Resolution* and *Positive Factoring*, presented in Figure 1. In the figure, σ is the *most general unifier* of B and C, i.e., a variable substitution to terms that satisfies two properties: (1) after the substitution B and C are identical, i.e., $B\sigma = C\sigma$, and (2) σ is a most general substitution that satisfies (1).

$$\frac{A \vee B \qquad \neg C \vee D}{A\sigma \vee D\sigma} \qquad\qquad \frac{A \vee B \vee C}{A\sigma \vee C\sigma}$$

Figure 1: Binary Resolution and Positive Factoring

Theorem 1 *Binary Resolution and Positive Factoring yield a calculus that is* sound *and* complete*. This means that a set of clauses is inconsistent if and only if there is a finite series of clauses $C_1, C_2, \ldots, C_n = \square$, where $\square$ denotes the empty clause, such that each clause is either member of the initial clause set or is obtained as a conclusion of Binary Resolution or Positive Factoring with premises selected from preceding clauses.*

PROOF: See [7]. □

Linear resolution As Theorem 1 indicates, resolution captures logical entailment very well. However, finding a deduction of the empty clause to show inconsistency can be rather tedious as we are given no guidance as to what clauses should be resolved in what order. To address this, various selection strategies have been devised, among them *linear resolution*.

Linear resolution is motivated by the idea that if we add a clause to a set of clauses that is considered consistent, then we only have to check the interactions that the new clause can have with the rest. Hence, in the first step, we resolve the new clause with some other, and in all subsequent steps, one of the premises will be the conclusion of the preceding step. Unfortunately, while in linear resolution the number of possible deductions is greatly decreased, we lose completeness. However, linear resolution remains complete for a restricted type of clauses that contain at most one positive literal, called *Horn clauses*. Besides, as it is shown in [4], linear resolution can be extended with a technique called *ancestor resolution* (see below in Subsection 2.3) which yields a complete calculus for the whole of First-Order Logic.

2.2 Programming in Prolog

Prolog [6] is a declarative programming language equipped with a built-in logical inference mechanism that corresponds to linear resolution. This mechanism is complete for Horn clauses, which correspond directly to Prolog rules. A rule has three parts: a head containing the only positive literal, the symbol *':-'* and a body which is the list of negative literals without negation, separated by commas. So, for instance, the Horn clause $P(X) \vee \neg Q_1(X) \vee \neg R(X,Y) \vee \neg Q_2(Y)$ corresponds to the Prolog rule

$$\texttt{P(X)} :- \texttt{Q}_1\texttt{(X)},\ \texttt{R(X,Y)},\ \texttt{Q}_2\texttt{(Y)}.$$

The semantics of this rule is as follows: if all atoms in the body are true, then so is the atom in the head. A Prolog program is a set of rules that can be used to prove a query atom, called *goal*. The program will try to unify the goal with some rule head, and in case of a successful unification, it will recursively try to prove each statement in the body. If the goal matches more than one rule head, then the program stores this by creating a so called *choice point* and proceeds with the first matching rule. If we manage to unify the goal with a bodiless rule head, then the goal is proved. If the inference fails, because there is no matching rule head, then we roll back to the last choice point and proceed with the next matching rule. This algorithm corresponds to linear resolution that starts from the negation of the query and that is always resolved in its first literal. This mechanism is very efficient in that it starts out from the goal and examines only those rules that have a potential to answer it.

2.3 Prolog Technology Theorem Proving

The Prolog Technology Theorem Prover approach (PTTP) was developed by Mark E. Stickel in the late 1980's [8]. PTTP is a sound and complete first-order theorem prover, built on top of Prolog. An arbitrary set of general clauses is transformed into a set of Horn-clauses that correspond to Prolog rules. Prolog execution on these rules yields first-order logic reasoning.

In PTTP, to each first-order clause we assign a set of Horn-clauses, the so-called *contrapositives*. The first-order clause $L_1 \vee L_2 \vee L_3 \vee \cdots \vee L_n$ has n contrapositives of the form

$L_k \leftarrow \neg L_1, \ldots, \neg L_{k-1}, \neg L_{k+1}, \ldots, \neg L_n$, for each $1 \leq k \leq n$. Having removed double negations, the remaining negations are eliminated by introducing new predicate names for negated literals. For each predicate name P a new predicate name *not_P* is introduced, and all occurrences of $\neg P(X)$ are replaced with *not_P*(X), both in the head and in the body. The link between the separate predicates P and *not_P* is provided using *ancestor resolution*, see below. For example, the clause $A(X) \vee \neg B(X) \vee \neg R(X,Y)$ is translated into three Prolog rules, each with different rule head:

```
A(X)       :-  B(X), R(X,Y).
not_B(X)   :-  not_A(X), R(X,Y).
not_R(X,Y) :-  not_A(X), B(X).
```

Thanks to using contrapositives, each literal of a first-order clause appears in the head of a Horn clause. This ensures that each literal can participate in a resolution step, in spite of the restricted selection rule of Prolog.

Next, let us see how PTTP implements positive factoring. Suppose we want to prove the goal A and during execution we obtain the subgoal $\neg A$. What this means that by this time we have inferred a rule, according to which if a series of goals starting with $\neg A$ is true, then A follows:

$$\texttt{A} \leftarrow \texttt{not\_A}, \texttt{P}_1, \texttt{P2}, \ldots \texttt{P}_k.$$

The logically equivalent first-order clause is

$$A \vee A \vee \neg P_1 \vee \neg P2 \vee \cdots \vee \neg P_k$$

from which we see immediately that the two occurrences of A can be unified, so there is no need to prove the subgoal `not_A`. This step is called *ancestor resolution* [4], which corresponds to the positive factoring inference rule. Ancestor resolution is implemented in Prolog by building an *ancestor list* which contains *open* predicate calls (i.e. goals that we started but have not yet finished proving).

Ancestor resolution is the inference step that checks if the ancestor list contains a goal which can be matched with the negation of the current goal. If this is the case, then the current goal succeeds and the unification with the ancestor element is performed. Note that in order to retain completeness, as an alternative to ancestor resolution, one has to try to prove the current goal using normal resolution, too. This is important if the ancestor element contains variables and a different proof can yield a different variable substitution.

There are two further features in the PTTP approach. First, to avoid infinite loops, iterative deepening is used instead of the standard depth-first Prolog search strategy. Second, in contrast with most Prolog systems, PTTP uses occurs check during unification, i.e., for example terms X and $f(X)$ are not allowed to be unified because this would result in a term of infinite depth.

To sum up, PTTP uses five techniques to build a first-order theorem prover on the top of Prolog: contrapositives, renaming of negated literals, ancestor resolution, iterative deepening, and occurs check.

2.4 DLog, a Description Logic Reasoner

The system DLog [5] is a Description Logic (DL) [1] reasoner for the $\mathcal{SHIQ}$ DL language, geared towards data reasoning, i.e., so called ABox inference. It proceeds by transforming the initial

knowledge base into a set of first-order clauses and then uses Prolog to perform the rest of the reasoning in a way similar to PTTP. The input of DLog is a proper sublanguage of First-Order Logic, hence the general PTTP approach can be optimised and simplified in a number of ways. Here we only highlight one difference between DLog and PTTP, that is relevant to the aims of the paper.

PTTP uses iterative deepening, i.e., traverses all branches of the search space in parallel. This is required due to the undecidability of First-Order Logic. In DLog, however, the only way we can obtain infinitely long branches is if we run into a loop, i.e., we try to prove the same goal that we started proving earlier. DLog detects and eliminates loops and hence is guaranteed to terminate for all input. Accordingly, DLog uses the standard depth-first search strategy of Prolog, which gives much better performance than iterative deepening.

3 Loop Elimination

In this section we present *loop elimination*, an important optimisation technique for both PTTP and DLog. Although both systems employ this optimisation, there has not yet been any rigorous proof of its soundness. In Subsection 3.1 we describe *proof trees* that can be used to represent Prolog execution. Afterwards, Subsection 3.2 constains the proof of soundness.

Definition 2 (Loop elimination) *Let P be a Prolog program and G a Prolog goal. Executing G w.r.t. P using* loop elimination *means the Prolog execution of G extended in the following way: we stop the given execution branch with a failure whenever we encounter a goal H that is identical to an open subgoal (that we started, but have not yet finished proving). Two goals are identical only if they contain the same variables, i.e., they are syntactically the same.*

Loop elimination is very intuitive. If, for example, we want to prove goal G and at some point we realise that this involves proving the same goal G, then there is no point in going further, because 1) either we fall in an infinite loop and obtain no proof or 2) we manage to prove the second occurrence of G in some other way that can be directly used to prove the first occurrence of the goal G. Things get complicated, however, due to ancestor resolution. The two G goals have different ancestor lists and it can be the case that we only manage to prove the second G due to the ancestors that the first G does not have. As it will turn out in the rest of this section, while we can indeed construct a proof of the first G from that of the second, this proof might have to be very different from the original one.

Although loop elimination is important for both PTTP and DLog, it has different role in the two systems. PTTP is a semi decision procedure for First-Order Logic and has to cope with infinite execution branches whether with or without loop elimination, hence it must use iterative deepening. Loop elimination helps to increase performance, but the basic operation remains the same. In DLog, however, loop elimination can itself ensure termination, because no branch of the proof tree can grow beyond any limit. As a result, DLog replaces iterative deepening search with depth-first search, which is much more efficient, since iterative deepening traverses the upper part of the tree many times.

3.1 Proof Trees

In this subsection we introduce *proof trees*, that are used to represent Prolog execution. We will only consider trees in the context of a PTTP like Prolog program, more precisely we will assume

that the program contains all contrapositives. Each tree node has a unique name and is labelled with a goal: `(Name:Goal)` refers to a node called `Name` and labelled with goal `Goal`. The root is labelled with the initial goal to be proved. Suppose the current goal G is unified with the head of rule

$$\mathtt{G} :- \quad \mathtt{B_1}, \mathtt{B_2}, \ldots, \mathtt{B_k}.$$

In this case, the node labelled G will have k children, each labelled $B1, B2, \ldots, Bk$, respectively. In each inference step, the validity of a goal is reduced to the validity of a set of goals in the children. After a successful execution, we obtain a proof tree such that each of its leaves can be considered true without further proof. We formalise this in the following definitions.

Definition 3 *An* atomic proof tree *consists of a root node labelled $A\sigma$ with children labelled $B1\sigma, B2\sigma, \ldots, Bn\sigma$, where σ is a variable substitution. We say that the atomic proof tree is* valid *if the corresponding Prolog program contains a rule*

$$\mathtt{A} :- \quad \mathtt{B_1}, \mathtt{B_2}, \ldots, \mathtt{B_n}.$$

A valid atomic proof tree can be seen as an instance of a rule. A proof tree *is built from atomic proof trees by matching nodes of identical labels. A proof tree is* valid *if all constituting atomic proof trees are valid.*

Remark 4 *The labels of proof trees are atomic predicates that can contain variables. Note that labels $p(X)$ and $p(Y)$ are not identical.*

Definition 5 *In a valid proof tree, a node labelled A is called* complete *if either 1) A can be unified with the head of a bodiless Prolog rule or 2) the node has an ancestor labelled $\neg A$ (ancestor resolution). A valid proof tree is complete if all its leafs are complete.*

To each successful Prolog execution that employs ancestor resolution, we can assign a complete proof tree.* In fact, the execution mechanism can be seen as a search in the space of complete proof trees. While standard Prolog will not necessarily traverse the whole space (because it might fall into an infinite loop), both PTTP and DLog are built so that they can enumerate all complete proof trees. This means that it is enough to show the existence of a complete proof tree to guarantee a successful PTTP or DLog execution.

Definition 6 *For an arbitrary child b of an atomic proof tree, the transformation* flipping over along the b child *is defined as follows: the root node is switched with its child b and their labels are negated. The rest of the tree is unaltered. This transformation is illustrated in Figure 2.*

Lemma 7 *For every valid atomic proof tree, the atomic tree obtained after flipping over along a child results in a valid atomic proof tree.*

PROOF: Let T be an atomic proof tree with root label A and children labelled $B, C_1, \ldots, C_k$. T is an instance of the Prolog clause

$$\mathtt{A} :- \quad \mathtt{B}, \mathtt{C_1}, \ldots, \mathtt{C_k}.$$

*In the Logic Programming community, it is customary to reserve the name proof tree only for complete proof trees. We introduce the notion of completeness because we will have to refer to trees that are not fully expanded.

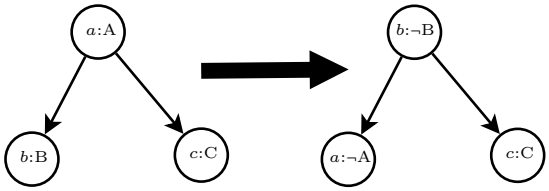

Figure 2: Flipping over along the b child

which is one of the contrapositives of the first-order clause $A \vee \neg B \vee \neg C_1 \vee \cdots \vee C_k$. Since the Prolog program contains all contrapositives of this clause, we also have

$$\text{not\_B} :- \quad \text{not\_A}, \mathtt{C_1}, \ldots, \mathtt{C_k}.$$

an instance of which corresponds to the flipped over version of T. □

Note that flipping over allows us to move between contrapositives of the same first-order clause.

Definition 8 *The transformation* flipping over along the $a, \bar{a}$ branch *is defined on proof trees as follows: let F be a proof tree, with node $(a : A)$ which has a leaf descendant $(\bar{a} : \neg A)$. In the branch leading to $\bar{a}$, let the nodes be $a = x_0, x_1, \ldots, x_{n-1}, x_n = \bar{a}$. To this tree we assign a tree F' which differs from F only in the subtree rooted at a. This subtree contains a branch $y_0 = x_n, y_1 = x_{n-1}, \ldots, y_i = x_{n-i}, \ldots, y_n = x_0$, and the label of each of these nodes is negated. Furthermore, each y_i in F' has the same siblings as x_{n-i+1} in F. The subtrees under the siblings are left unaltered. This transformation is illustrated in Figure 3.*

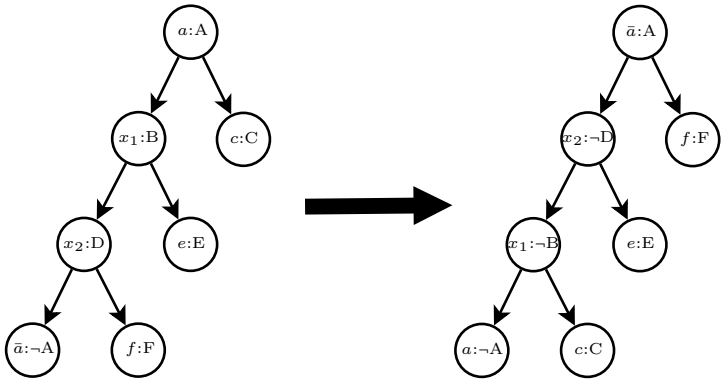

Figure 3: Flipping over along the $(a, \bar{a})$ branch

Lemma 9 *If we have a complete proof tree T that contains nodes $(a : A)$ and $(\bar{a} : \neg A)$ such that $\bar{a}$ is a leaf descendant of a, then the tree obtained after flipping T along the $(a, \bar{a})$ branch is a valid proof tree.*

PROOF: The new downward path $\bar{a} \to a$ consists of atomic trees that are the flipped over versions of the atomic trees of the initial upward path $\bar{a} \to a$. We know from Lemma 7 that flipping over a valid atomic proof tree yields another valid atomic proof tree, hence the whole new proof tree is valid. □

Remark 10 *Although we obtained a valid proof tree after flipping over, the proof tree is not necessarily complete. This is because some ancestor lists change and branches that previously terminated in ancestor resolution might have to be expanded further (because the required ancestor disappeared).*

3.2 The Soundness of Loop Elimination

In this subsection we show that for every complete proof tree that contains loops, one can construct a complete proof tree that is loop free.

Definition 11 *Let F be a complete proof tree containing a loop L, i.e. a pair of nodes $(p_1 : P), (p_2 : P)$, for some label P, such that p_2 is a descendant of p_1. We define the* depth *of L to be the distance of p_1 from the root.*

Definition 12 *A node $n : N$ is said to be* eligible for ancestor resolution *if it has an ancestor with label $\neg N$. If an inner node is eligible for ancestor resolution, then it is called a* bad node.

Bad nodes are called bad, because they are unnecessarily expanded. There is no need to provide a proof tree under a bad node, since it is complete even if it remains a leaf.

Lemma 13 *If we have a complete proof tree that contains a bad node n, then the tree obtained after removing the subtree under n yields a complete proof tree in which n is not bad any more.*

PROOF: Removing the subtree under n makes n a leaf node. However, n is complete due to ancestor resolution. The rest of the leaves are unaltered, so they remain complete. Hence, the new proof tree is complete. □

Lemma 14 *Let F be a complete proof tree containing a loop L at depth d. It is possible to find another complete proof tree F' for the same goal (i.e., with the same label in the root) such that F' can contain the loops of F except for L, plus possibly loops that have depth strictly greater than d.*

PROOF: Let $(p_1 : P)$ and $(p_2 : P)$ be the first and second nodes of the loop. First, we eliminate all bad nodes by removing the subtrees rooted at these nodes. According to Lemma 13, we obtain a complete proof tree. In this tree, all nodes that are eligible for ancestor resolution are leaf nodes. The ancestor list of p_2 contains the ancestors of p_1 plus the nodes on the path between p_1 and p_2. Let ANC denote the set of nodes between p_1 and p_2.

In case none of the nodes in ANC play any role in the proof of p_2 (i.e., they do not participate in ancestor resolution), the proof of p_1 can be directly replaced with that of p_2, eliminating loop L. This is illustrated in Figure 4. We obtain a complete proof tree that satisfies our lemma.

The situation is more complicated when some nodes in ANC participate in ancestor resolution under p_2. Among these, let $(a : A)$ be the lowest one (i.e., the last one to enter the ancestor list). Somewhere under p_2 there is a leaf $(\bar{a} : \neg A)$ that is complete due to ancestor resolution. Let us flip over F along the branch $(a, \bar{a})$. In the flipped over branch the nodes between a and $\bar{a}$ will appear with negated labels and in inverse order. Afterwards, we once more eliminate all bad nodes by removing the subtrees under them. p_2 is on the path between a and $\bar{a}$, so its

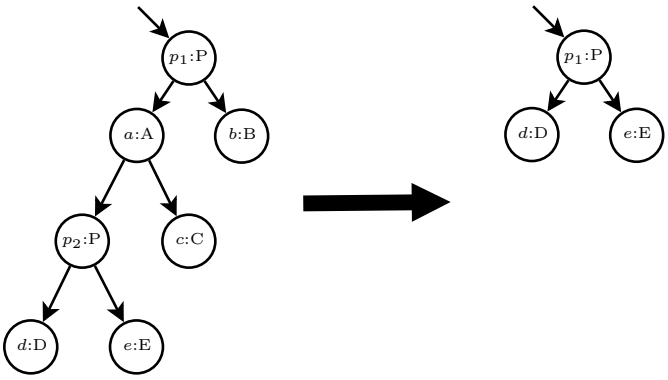

Figure 4: Replacing the proof of p_1 with that of p_2

label will turn to $\neg P$, which makes p_2 eligible for ancestor resolution. Hence, when we eliminate badness, we eliminate the subtree under p_2. As a result, loop L disappears. An example of this is shown in Figure 5. We know that flipping a complete proof tree results in a valid proof tree, but it is not necessarily complete, because some goals that previously succeeded with ancestor resolution might loose the required ancestor (cf. Remark 10). This is the case when there is a node $(b : B)$ under a and somewhere underneath there is a leaf $(\bar{b} : \neg B)$. Node b has to be on the path between a and $\bar{a}$ otherwise b will continue to be an ancestor of $\bar{b}$ and their labels will not change. There are two possibilities:

1. As it is illustrated in Figure 6, b lies between a and p_2. Then, $\bar{b}$ cannot appear under p_2, because a was chosen to be the lowest such node. Hence, $\bar{b}$ appears under b, but not under p_2. After flipping, both b and $\bar{b}$ will appear under p_2, so they will be eliminated when we eliminate the badness of p_2. Hence, this case will not yield any incomplete leaves.

2. We illustrate the second case, namely when b is under p_2 in Figure 7. We will treat all such nodes together, i.e., let $(b_1 : B_1), (b_2 : B_2), \ldots (b_k : B_k)$ be nodes on the path between p_2 and $\bar{a}$ (nodes b, c on Figure 7), such that each b_i has at least one leaf descendant $(\bar{b}_{il} : \neg B_i)$. The nodes are ordered so that b_1 is the closest to p_2 and b_k is the farthest. After flipping over, the labels of these nodes will be negated, i.e., turn to $\neg B_i$, respectively, and they will

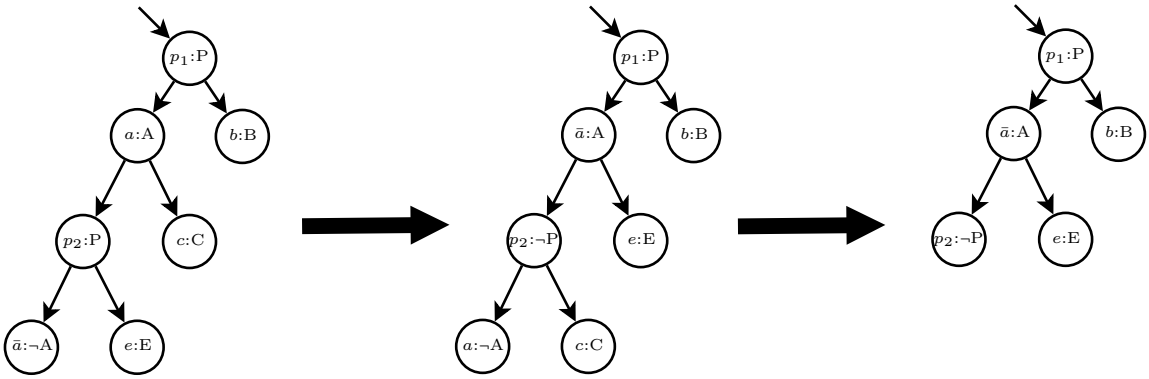

Figure 5: Flipping over along the $(a, \bar{a})$ branch, then bad node elimination

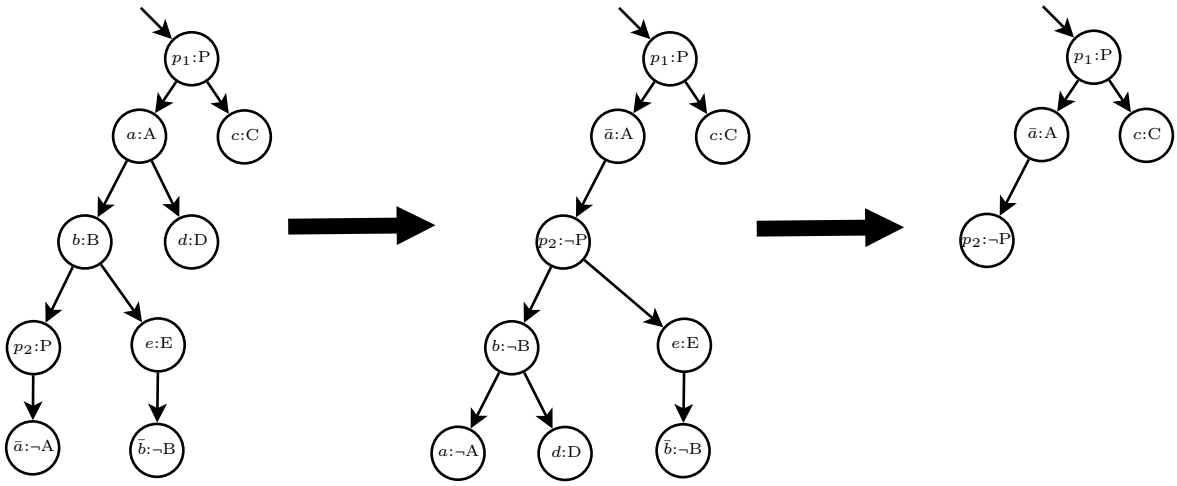

Figure 6: Ancestor resolution eliminates both b and $\bar{b}$

appear on the branch leading to p_2 in inverted order, i.e., b_k will be the topmost, while b_1 the lowest.

Let us consider b_1. Due to flipping over, it will lose all its previous descendants. Its new descendants will be its previous ancestors on the path between p_2 and b_1 along with their descendants towards other branches. We claim that none of the new descendants of b_1 can have lost an ancestor which previously allowed for ancestor resolution, i.e., none can be one of $\bar{b}_{il}$. This is because the lost ancestor would have been above b_1, however, b_1 was chosen to be the topmost one. Consequently, the subtree under b_1 after flipping has no incomplete leaves, i.e., the whole subtree is complete. The label of b_1 is $\neg B_1$, so we have a complete proof for $\neg B_1$. This means that we can copy the subtree under b_1 to any node $(\bar{b}_{1l} : \neg B_1)$, thus compensating such nodes for the lost ancestor. Note that we need to rename the copied nodes to ensure that each node has a unique name.

We next turn to b_2. Through analogous reasoning we can see that the new leaf descendants of b_2 are either complete or else are incomplete because they lost an ancestor labelled $\neg B_1$. However, by copying the subtree under b_1, we have already turned such leaves into complete trees. Hence, we have a complete proof tree under b_2, proving $\neg B_2$, which we copy to any incomplete leaf $(\bar{b}_{2l} : \neg B_2)$ (again assigning new names to the newly created nodes).

We continue the process. In the i^{th} step, we have a complete proof tree under b_i which we copy to any leaf $(\bar{b}_{il} : \neg B_i)$. By the end of the k^{th} step, the we obtain a complete proof tree. Note that we make exactly one copying for each leaf $\bar{b}_{il}$ that lost its completeness after flipping over, so copying terminates.

Flipping over turns the label of p_2 from P to $\neg P$, which makes loop L disappear. New loops can arise (some nodes were negated), however, no such loop can start above p_1. We show this by contradiction. Suppose a node $(n_1 : N)$ above p_1 obtains a descendant $(n_2 : N)$ after flipping. The labels of the nodes under n_1 in the new tree are either the same or the negated labels that appeared under n_1 before flipping. So, if a new loop appeared, it was because the label of a descendant of n_1, namely of n_2, changed from $\neg N$ to N. This means that before flipping over n_2 was eligible for ancestor resolution. Since we eliminated all

bad nodes, n_2 was a leaf. However, flipping over does not negate the labels of leaf nodes, so we obtained a contradiction.

We conclude that the possibly arising loops are all of greater depth than the eliminated loop. □

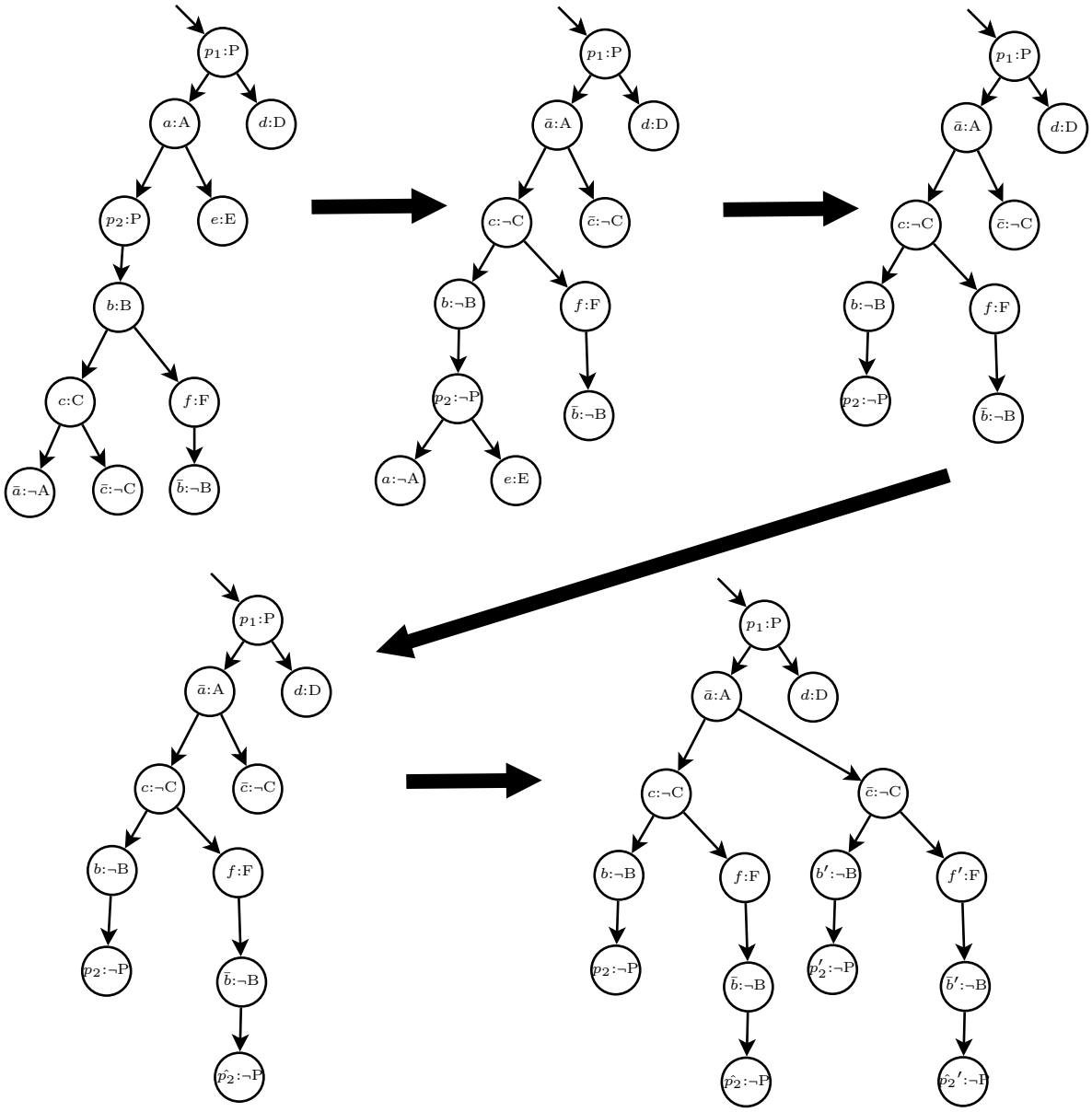

Figure 7: Copying makes first $\bar{b}$, then $\bar{c}$ complete

Theorem 15 *For every complete proof tree containing loops there is a complete proof tree that is loop free.*

Proof: Using Lemma 14, we can eliminate any loop at the expenses of possibly creating some loops that have greater depth. Let us eliminate loops in order of increasing depth. After each

iteration, we know that loops are getting deeper and deeper. There are two possibilities:

1. Eventually, we manage to eliminate each loop after a finite number of iterations. The resulting proof tree satisfies our theorem.

2. The elimination never terminates. Since the loops are getting farther from the root, it follows that the part of the proof tree that is loop free grows beyond any limit. Suppose the initial tree contains n distinct labels in its nodes. The transformation steps involve flipping over, copying subtrees and eliminating nodes, each of which either preserves node labels or introduces the negation of some label to a node. Hence, there can be at most $2n$ distinct labels, i.e., any loop free path from the root node can be at most $2n$ long. This contradicts the assumption that the loop free part of the tree grows beyond any limit. Hence, all loops have to disappear after finitely many iterations.

□

4 Conclusion

Prolog based inference systems like PTTP and DLog can be used to prove a query goal. We have shown is Section 3 that these systems need not explore proof trees that contain loops, because in case there is a complete proof tree, there is one without loops (Theorem 15). This allows for reducing the search space, making both systems faster. Besides, loop elimination is sufficient to make the DLog reasoner terminating, thus allowing to replace iterative deepening search with depth-first search, which further increases performance.

References

[1] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2004.

[2] R. N. Bol, K. R. Apt, and J. W. Klop. An analysis of loop checking mechanisms for logic programs. *Theor. Comput. Sci.*, 86:35–79, August 1991.

[3] M. Fitting. *First-order logic and automated theorem proving.* Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[4] R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[5] G. Lukácsy and P. Szeredi. Efficient description logic reasoning in prolog: The DLog system. *Theory and Practice of Logic Programming*, 9(03):343–414, 2009.

[6] ISO Prolog standard, 1995. ISO/IEC 13211-1.

[7] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[8] M. E. Stickel. A Prolog technology theorem prover: a new exposition and implementation in Prolog. *Theoretical Computer Science*, 104(1):109–128, 1992.