Acta Cybernetica **20** (2012) 441–458.

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

Zsolt Zombori and Péter Szeredi†

Abstract

In this paper we present *loop elimination*, an important optimisation technique for first-order theorem proving based on Prolog technology, such as the Prolog Technology Theorem Prover or the DLog Description Logic Reasoner. 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 [8] 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 [7] 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) [9] and the DLog system [6] 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, which can make a tremendous impact on the speed of both of the aforementioned systems. This technique prevents logic programs from trying to prove the same goal over and over again, thus avoiding certain types of infinite loops.

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

*The work reported in the paper has been developed in the framework of the project "Talent care and cultivation in the scientific workshops of BME" project. This project is supported by the grant TÁMOP - 4.2.2.B-10/1–2010-0009

†Budapest University of Technology and Econcomics, Department of Comptuer Science and Information Theory, Budapest, Magyar tudósok körútja 2. H-1117, Hungary, E-mail: {zombori,szeredi}@cs.bme.hu

loop elimination do not hold. It is easy to see that trying to prove a goal that is identical to some goal that we are already in the process of proving yields no useful solution and the corresponding proof attempt can be aborted. However, it is far from trivial that the same holds in case the two goals are identical only *modulo ancestor list*, i.e., they can be different in one of their arguments, namely in their list of ancestors. Our paper proves this stronger claim. We are not aware of any other work exploring the interaction between loop elimination and ancestor resolution.

In Section 2 we provide an overview of resolution reasoning and Prolog programming, that will be necessary for understanding the rest of the paper. In Section 3 we examine logic programs in terms of termination and identify the sources of infinite execution. Section 4 contains our main contribution: we define loop elimination and prove its soundness. We end the paper with some concluding remarks in Section 5.

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. We expect the reader to be familiar with the basics of First-Order Logic.

2.1 Resolution Theorem Proving

Resolution [8] 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, the clauses above the bar are the premises of the inference and the clause under the bar is the conclusion. 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). In Figure 2, we illustrate the

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

Figure 1: Binary Resolution and Positive Factoring

application of the the two inference rules. On the left side the Binary Resolution rule is

used and on the right side the Positive Factoring rule fires. The most general unifier is the same in both examples: variable y is mapped to x and every other variable is mapped to itself.

$$\frac{A(x)\vee B(x) \qquad \neg B(y)\vee D(y)}{A(x)\vee D(x)} \qquad\qquad \frac{A(x)\vee B(x)\vee B(y)\vee D(y)}{A(x)\vee B(x)\vee D(x)}$$

Figure 2: Examples illustrating the Binary Resolution and Positive Factoring inference rules

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 a member of the initial clause set or is obtained as a conclusion of Binary Resolution or Positive Factoring with premises selected from preceding clauses.*

A proof of Theorem 1 can be found, for example, in [8].

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 [5], 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 [7] 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

```
P(X) :- Q1(X), R(X,Y), Q2(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 remembers 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 [9]. 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:

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

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* [5], 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 [6] 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 performs a two-phase reasoning. The first phase deals only with the so called terminology box (TBox) part of the knowledge base that contains general background knowledge. The point of this phase is that by the end we obtain a syntactically simpler set of clauses to be used in the subsequent data reasoning. The second phase uses Prolog to perform the rest of the reasoning in a way similar to PTTP. Due to the syntactically simpler input clause set, the general PTTP approach can be optimised and simplified in a number of ways.

In [10] we describe the algorithm of the first phase, as a result of which we obtain clauses of the following types:

$$\neg R(x,y) \vee S(y,x) \tag{1}$$

$$\neg R(x,y) \vee S(x,y) \tag{2}$$

$$\mathbf{P}(x) \tag{3}$$

$$\bigvee_i \left(\mathbf{P_1}(x_i) \vee \left(\bigvee_j \neg R(x_i,y_j) \vee \bigvee_j \mathbf{P_2}(y_j) \vee \bigvee_{j_1,j_2} (y_{j_1} = y_{j_2}) \right) \right) \tag{4}$$

$$R(a,b) \tag{5}$$

$$C(a) \tag{6}$$

$$a = b \tag{7}$$

$$a \neq b \tag{8}$$

where $\mathbf{P}(x)$ is a shorthand for $(\neg)P_1(x) \vee (\neg)P_2(x) \vee \cdots \vee (\neg)P_k(x)$. Note some nice properties of our clause set:

1. There are no function symbols.
2. In the contrapositives generated from these clauses a negative binary literal can only appear in the body in case the head is a negative binary literal (clauses of type 1 and 2).
3. For every clause that contains binary literals, all variables occur in some binary literal.
4. Clauses that do not contain binary literals have at most one variable.

As we will see later, these nice properties allow for specializing PTTP to obtain a terminating decision procedure for the $\mathcal{SHIQ}$ DL language.

3 Termination of Logic Programs

Given that first-order logic is undecidable, it is not surprising that the Prolog Technology Theorem Prover is not guaranteed to terminate. In this section we review the ways in which a logic program can fall short of termination. Afterwards, we compare PTTP and DLog with respect to termination.

3.1 Sources of infinite execution

We identify three sources of infinite execution:

- If the program contains **function symbols**, then we might obtain terms of ever increasing depth. Consider, for example, the following simple program:

  ```
  p(X)      :-  p(f(X)).
  ```

  If we attempt to prove $p(a)$ using the above rule, we will end up reducing it to the proof of $p(f(a))$, $p(f(f(a)))$ etc. and the program will never stop.

- A proof attempt might visit infinitely many goals if an unbounded number of **new variables** can be introduced during the proof. This is the case for example with the transitivity rule:

  ```
  r(X,Y)    :-  r(X,Z), r(Z,Y).
  ```

  It is easy to see that a proof attempt for the goal $r(a,b)$ using the above rule will generate infinitely many $r(X,Y)$ subgoals, always with fresh variables.

- Even if both the depth of terms and the number of variables can be bounded, the program might fall into a **loop** and attempt to prove the same goal over and over again. For example, the program consisting of the following rule

```
p(X)        :-  p(X).
```

will never terminate, even though there are no function symbols and no new variables are introduced.

One can see easily that the above list is exhaustive. If the number of variables is bounded and there are no functions, then the total set of terms is that of the variables and the constants appearing in the program, i.e., it is finite. Since the predicate names are also finite, there can be finitely many different goals. If there are no loops, even if a proof attempt goes though all possible goals (the worst case), it will eventually terminate.

Hence, we conclude that infinite execution is due exactly to three aspects of logic progams: function symbols, the proliferation of new variables and loops.

3.2 Termination in DLog

In light of the preceding subsection, let us reexamine the input clause set of the DLog data reasoner. We see immediately that the absence of function symbols eliminates one of the three sources of infinite execution.

We shall see that new variables are not introduced, either. The second nice property of the input clause set is that the resulting contrapositives only contain a negative binary literal in the body in case the head is a negative binary literal. This means that we can only encounter negative binary subgoals if the initial query itself is a negative binary goal. In $\mathcal{SHIQ}$ DL reasoning, however, negative binary queries are forbidden, so all contrapositives with a negative binary literal are unnecessary and can be disposed of. Consequently, in our logic program binary literals will only appear positively. For proving such binary goals only contrapositives from clauses of type 1 and 2 are available:

```
r(X,Y)      :-  s(X,Y).
r(X,Y)      :-  s(Y,X).
```

These rules do not introduce new variables. A proof of a binary goal consists of applying such rules possibly several times, until finally we obtain a matching data assertion `r(a,b)`, thanks to which the variables in the binary goal get instantiated. We know that in all rule bodies that contain binary literals every variable occurs in some binary literal (the third nice property of our input clause set). These are the rules that introduce new variables. If, however, we move the binary literals to the front of the body, i.e., we prove the binary goals first, by the time we reach the unary goals, they become ground. Hence, any unary goal in the body either contains the same variable as the one in the head – in case the rule contains no binary predicates – or else it is ground by the time it is called. New variables may appear only for a short time – until we prove the binary goals holding them. Hence, DLog will never encounter infinitely many new variables during a proof attempt.

If there are no terms of increasing depth and variables do not proliferate, then the only way a DLog program may not terminate is if it falls in an infinite loop and proves the same goal repeatedly.

3.3 Eliminating Loops

We have seen that there are three independent features that can make a PTTP execution non-terminating, of which only one, namely loops can occur in DLog programs. In Section 4 we shall show that proofs containing such loops are not necessary for completeness. This result yields an important optimization for both PTTP and DLog, called *loop elimination*. General PTTP still has to cope with infinite proof attempts (due to the other two sources) and hence has to use iterative deepening, i.e., build several proof attempts in parallel. However, even if loop elimination does not allow for changing the proof search strategy, but it still prunes the search space significantly. In DLog, loop elimination eliminates the only remaining source of infinite proofs. Accordingly, DLog always terminates and uses the standard depth-first search strategy of Prolog, which gives much better performance than iterative deepening.

4 Loop Elimination

In this section we present the optimization heuristic *loop elimination* for both PTTP and DLog. In the literature, loop elimination is often referred to as *identical ancestor pruning*, see for example [9] or [4]. Although both systems employ this optimisation, there has not yet been any rigorous proof of its soundness. In Subsection 4.1 we describe *proof trees* that can be used to represent Prolog execution. Afterwards, Subsection 4.2 contains the proof of soundness.

Definition 1 (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 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. This is the standard justification that we find in the literature. For example [4] says:

> Identical ancestor pruning (IAP) is a powerful pruning heuristic in a model elimination search. Imagine, in the course of expanding a ME proof space for a particular goal P, that one were to encounter that same goal P again. One of two situations must hold:
>
> 1. There are no proofs of P from this database (because it doesn't logically follow).
> 2. Whether or not there is a proof using this second occurrence of P, there must be another proof of the original P not using it. Also, the different proof occurs at a shallower depth.

> This is true because the second occurrence must eventually be proven somehow, so this recursion must bottom out. And then, by whatever proof this second occurrence succeeds, an analogous proof path must exist below the first occurrence of P. In either case, it is justifiable to prune the space below the second occurrence of P.

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.

4.1 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 $B_1, B_2, \ldots, B_k$, 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 2. *An* atomic proof tree *consists of a root node labelled $A\sigma$ with children labelled $B_1\sigma, B_2\sigma, \ldots, B_n\sigma$, 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 1. The labels of proof trees are atomic predicates that can contain variables. Note that labels $p(X)$ and $p(Y)$ are not identical.

Definition 3. *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.[1] 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 4. *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 3.*

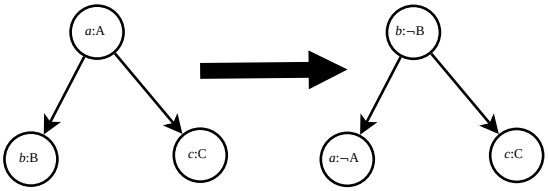

Figure 3: Flipping over along the *b* child

Lemma 1. *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 the root node labelled $A\sigma$ and children labelled $B\sigma, C_1\sigma, \ldots, C_k\sigma$. T is an instance of the Prolog clause

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

which is a contrapositive 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

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

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 5. *The transformation* flipping over along the $a, \bar{a}$ branch *is defined on proof trees as follows: let F be a proof tree, with a node $(a : A)$ which has a leaf descendant $(\bar{a} : \neg A)$. The nodes on the path from a to $\bar{a}$ are $a = x_0, x_1, \ldots, x_{n-1}, x_n = \bar{a}$. 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 4.*

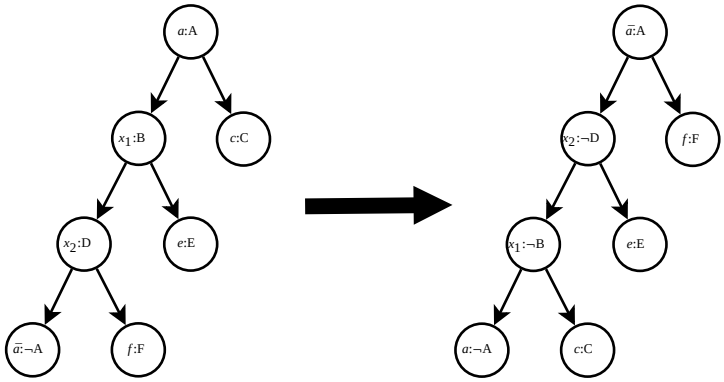

Figure 4: Flipping over along the $(a, \bar{a})$ branch

Lemma 2. *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 1 that flipping over a valid atomic proof tree yields another valid atomic proof tree, hence the whole new proof tree is valid. □

Remark 2. 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).

4.2 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 6. *A complete proof tree is said to contain a loop L if it contains a pair of nodes $(p_1 : P), (p_2 : P)$, for some label P, such that p_2 is a descendant of p_1. Node p_1 is called the* top node *and node p_2 the* bottom node *of the loop L. We define the* depth *of L to be the distance of p_1 from the root.*

Definition 7. *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.

[1] 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.

Lemma 3. *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. □

Definition 8. *We define the* loop-depth *of a tree T with a pair of integers $(-D,C)$, where D is the minimum depth of all loops in T and C is the number of nodes that are bottom nodes of some loop of depth D. If the tree contains no loops, then its loop-depth is $(-\infty,0)$. Loop-depths are comparable using lexicographic ordering, i.e., loop-depth (A,B) is less than loop-depth (C,D) if and only if either $A<C$ or else $A=C$ and $B<D$.*

Lemma 4. *Let F be a complete proof tree with loop-depth LD that contains at least one loop. 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 the loop-depth of F' is strictly less than LD.*

Proof. The loop-depth of F is LD=$(-D,C)$. This means that there is at least one loop of depth D and there are no loops with depth less than D. Let L be one such loop with top and bottom nodes $(p_1:P)$ and $(p_2:P)$, respectively. First, we eliminate all bad nodes by removing the subtrees rooted at the bad nodes. According to Lemma 3, we obtain a complete proof tree.

In case the elimination of the subtrees under bad nodes eliminates loop L, then the obtained complete proof tree has loop-depth $(-D_2,C_2)$. In case there were no other loops of depth D in F then $D_2>D$. Otherwise, $D_2=D$ and $C_2=C-1$. In either case $(-D_2,C_2)<(-D,C)$, so our lemma is satisfied.

Otherwise, in the obtained 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 5. We obtained a complete proof tree F' with loop-depth $(-D_2,C_2)$. In case there were no other loops of depth D in F then $D_2>D$. Otherwise, $D_2=D$ and $C_2=C-1$. In either case $(-D_2,C_2)<(-D,C)$, so our lemma is satisfied.

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. Node p_2 is on the path between a and $\bar{a}$, so its 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 6. We know that flipping a complete proof tree results in a valid proof tree, but it is not necessarily

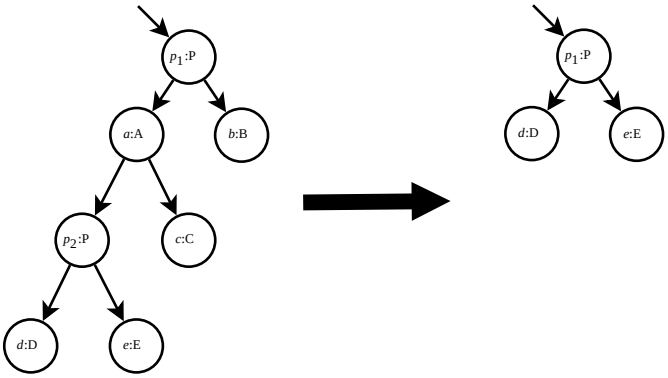

Figure 5: Replacing the proof of p_1 with that of p_2

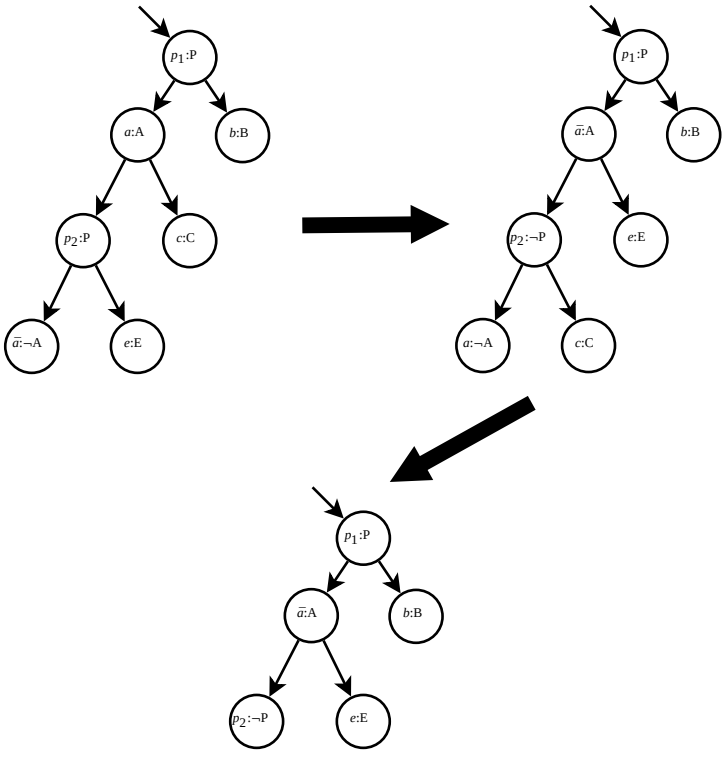

Figure 6: Flipping over along the $(a, \bar{a})$ branch, then bad node elimination

complete, because some goals that previously succeeded with ancestor resolution might loose the required ancestor (cf. Remark 2). This is the case when there is a node $(b:B)$ under a and somewhere underneath there is a leaf $(\overline{b}:\neg B)$. Node b has to be on the path between a and $\overline{a}$ otherwise b will continue to be an ancestor of $\overline{b}$ and their labels will not change. There are two possibilities:

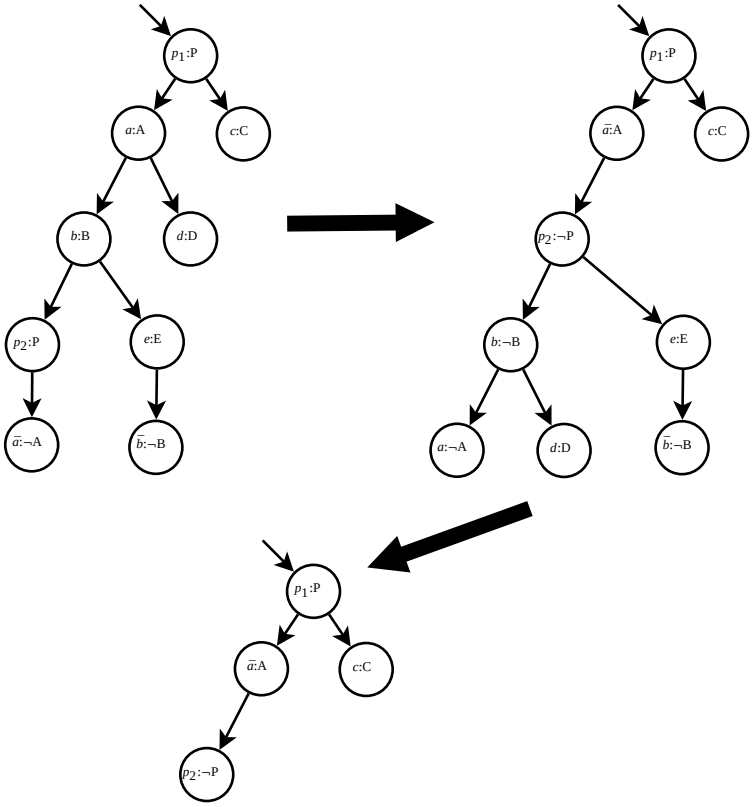

Figure 7: Ancestor resolution eliminates both b and $\overline{b}$

1. As it is illustrated in Figure 7, b lies between a and p_2. Then, $\overline{b}$ cannot appear under p_2, because a was chosen to be the lowest node participating in ancestor resolution under p_2. Hence, $\overline{b}$ appears under b, but not under p_2. After flipping, both b and $\overline{b}$ 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 8. 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 a and $\overline{a}$ (nodes b,c on Figure 8), such that each b_i has at least one leaf descendant $(\overline{b_{il}}:\neg B_i)$. 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 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 $\overline{a}$ncestor which previously allowed for ancestor resolution, i.e., none can be one of $\overline{b_{il}}$. 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. This subtree in itself is not necessarily complete, because the ancestors of $\overline{a}$ might be needed for some ancestor resolution steps. We express this by saying that the subtree under b_1 is *complete in the context of the ancestors of* $\overline{a}$. In the following, we will always assume the same context (the ancestors of $\overline{a}$) and will omit specifying it whenever it leads to no misunderstanding. The label of b_1 is $\neg B_1$, so we have a complete proof for $\neg B_1$ (again in the context of the ancestors of $\overline{a}$). This means that we can copy the subtree under b_1 to any node $(\overline{b_{1l}} : \neg B_1)$, 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 (in the context of $\overline{a}$), proving $\neg B_2$, which we copy to any incomplete leaf $(\overline{b_{2l}} : \neg B_2)$ (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 $(\overline{b_{il}} : \neg B_i)$. By the end of the k^{th} step, we obtain a complete proof tree. Note that we make exactly one copying for each leaf $\overline{b_{il}}$ that lost its completeness after flipping over, so copying terminates.

We now obtained a new proof tree F'. Let us show that F' has the properties claimed by the lemma being proved. 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 or at p_1. We show this by contradiction. Suppose a node $(n_1 : N)$ above or at 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 either because the bottom node of an already existing loop L_2 was copied or because the label of a descendant of n_1, namely of n_2, changed from $\neg N$ to N. In the first case, the depth of loop L_2 is smaller than the depth of loop L, which is impossible because L was chosen to be a loop of minimum depth (cf. Definition 8. of loop-depth). In the second case, 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. Hence, the number of loops of depth D is reduced by one, i.e., the loop-depth of the new tree is strictly less than that of the original tree. □

Theorem 2. *For every complete proof tree containing loops there is a complete proof tree that is loop free.*

Proof. Using the transformation described in Lemma 4, we can create a series of proof trees of the same goal such that the loop-depth is always decreasing. The second component of the loop-depth is a positive integer (the number of loops at minimum depth) which cannot decrease infinitely, so eventually the first component will decrease as well. This means that the minimum depth of the loops increases, i.e. loops get 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.

□

5 Conclusion

Prolog based inference systems like PTTP and DLog can be used to prove a query goal. We have shown is Section 4 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 2). This allows for reducing the search space, making both systems faster. Besides, loop elimination is sufficient to make the DLog reasoner terminating, thus allowing one to replace iterative deepening search with depth-first search, which further increases performance.

References

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

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

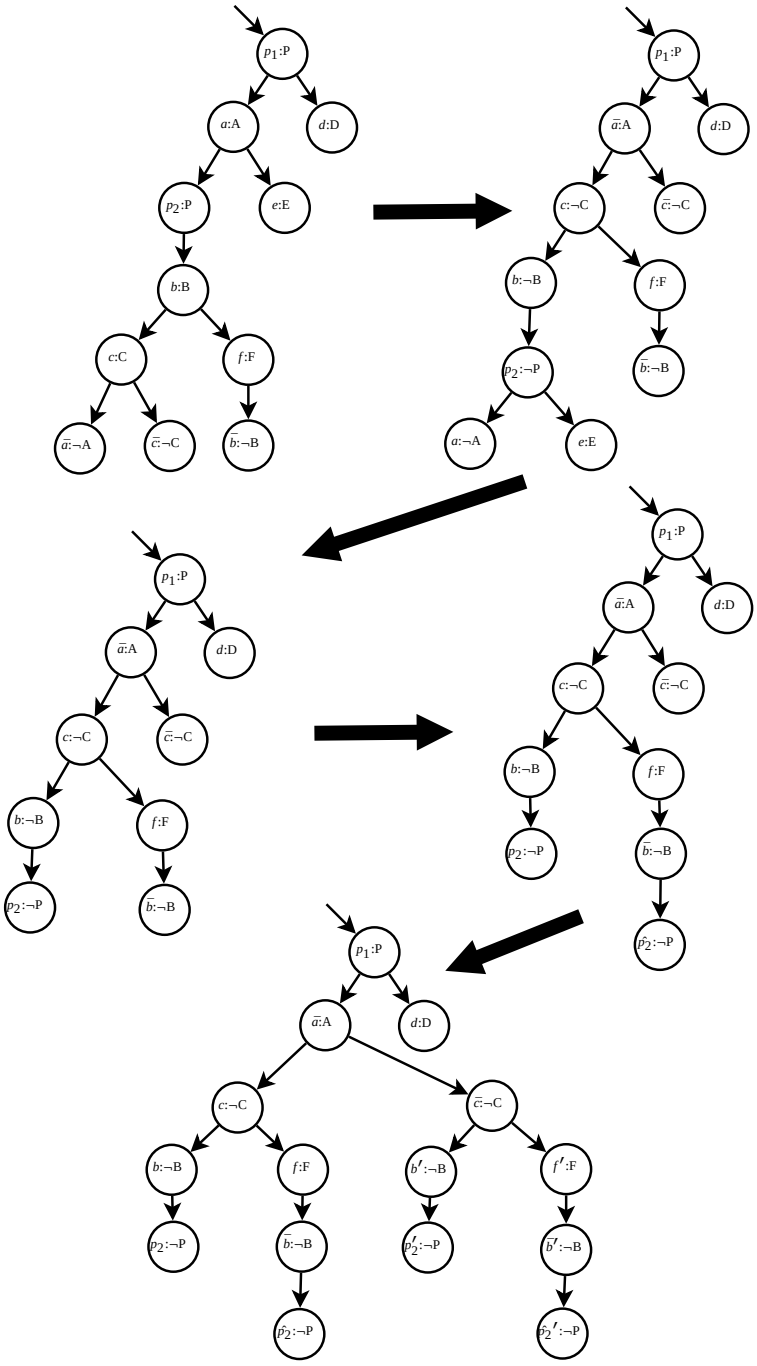

Figure 8: Copying makes first $\bar{b}$, then $\bar{c}$ complete

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

[4] Geddis, Donald F. *Caching and non-Horn Inference in Model Elimination Theorem Provers*. PhD thesis, Stanford University, USA, June 1995.

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

[6] Lukácsy, Gergely and Szeredi, Péter. Efficient Description Logic reasoning in Prolog: The DLog system. *Theory and Practice of Logic Programming*, 9(03):343–414, 2009.

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

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

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

[10] Zombori, Zsolt. Efficient two-phase data reasoning for Description Logics. In *IFIP AI*, pages 393–402, 2008.

Received 15th November 2011