On semantic resolution with lemmaizing and contraction and a formal treatment of caching

Maria Paola Bonacina *
Department of Computer Science
University of Iowa
Iowa City, IA 52242-1419, USA
bonacina@cs.uiowa.edu

Jieh Hsiang †
Department of Computer Science
National Taiwan University
Taipei, Taiwan
hsiang@csie.ntu.edu.tw

Abstract

Reducing redundancy in search has been a major concern for automated deduction. Subgoal-reduction strategies, such as those based on model elimination and implemented in Prolog technology theorem provers, prevent redundant search by using *lemmaizing* and *caching*, whereas contraction-based strategies prevent redundant search by using *contraction* rules, such as *subsumption*. In this work we show that lemmaizing and contraction can coexist in the framework of *semantic resolution*. On the lemmaizing side, we define two meta-level inference rules for lemmaizing in semantic resolution, one producing unit lemmas and one producing non-unit lemmas, and we prove their soundness. Rules for lemmaizing are meta-rules because they use global knowledge about the derivation, e.g. ancestry relations, in order to derive lemmas. Our meta-rules for lemmaizing generalize to semantic resolution the rules for lemmaizing in model elimination. On the contraction side, we give contraction rules for semantic strategies, and we define a *purity deletion* rule for first-order clauses that preserves completeness. While lemmaizing generalizes success caching of model elimination, purity deletion echoes failure caching. Thus, our approach integrates features of backward and forward reasoning. We also discuss the relevance of our work to logic programming.

1 Introduction

Some of the most successful theorem-proving programs existing today implement either contraction-based strategies (e.g., [2, 16, 23]) or subgoal-reduction strategies (e.g., [4, 19, 31]). These two classes of strategies represent two different approaches to refutational theorem proving.

Contraction-based strategies (e.g., [6, 13, 15, 28]) are *forward-reasoning* strategies, that prove the target theorem by deriving consequences from the axioms. (Most forward-reasoning resolution strategies do not differentiate between the target theorem (negated) and the axioms. They treat them as a single set of clauses and try to derive a contradiction from it.) Because generated clauses are kept, the strategy works on a database of clauses. The primary strength of the

*Supported in part by the National Science Foundation with grant CCR-94-08667.
†Supported in part by grant NSC 85-2221-E-002-009 of the National Science Council of the Republic of China.

contraction strategies, and also the reason for their name, is that they apply eagerly contraction inference rules, such as *simplification* and *subsumption*, to delete clauses that are not needed to prove the target theorem. We call these deleted clauses *redundant*. Contraction inference rules are defined based on well-founded orderings on terms and literals, so that by applying such rules the strategy keeps its database mimimal with respect to those orderings [8, 28]. By effectively reducing redundancy, contraction-based strategies keep the size of the database in check and have been used successfully to prove many problems beyond the reach of other types of strategies (e.g., [1, 2, 17, 24]). The orderings are also used to restrict the applicability of expansion inference rules, as in *ordered resolution* and *ordered paramodulation*. Because the strategies work on a database of clauses, they may feature a variety of *search plans* to control the selection of clauses for the inferences. Finally, these strategies employ *indexing* techniques [11, 22, 32] that allow them to execute very rapidly operations such as, given a term, retrieve all its instances, anti-instances or unifying terms in the database. The fast execution of these operations is fundamental to the practical success of these strategies, because they need to test the applicability of inference rules on large numbers of clauses.

The subgoal-reduction strategies are *linear*, *backward-reasoning* strategies. In such a strategy an inference step consists in reducing the current goal to a set of subgoals, starting from the input goal. Typical examples are the *Prolog technology theorem provers* [31], and *model elimination* [20]. The latter is also a fundamental methodology on which many such strategies are based. A weakness of a pure subgoal-reduction strategy is that by concentrating only on the current goal it has no memory of previously solved goals. Therefore, if the same subgoals, or instances thereof, are generated at different stages, the strategy solves them independently, repeating the same steps. More sophisticated subgoal-reduction strategies avoid such repetitions by using techniques of *lemmaizing*, that is, saving solved goals as lemmas. Lemmaizing for model elimination was presented already in [20]. However, its first implementation in [14] was less efficient than expected [4], because unrestricted lemmaizing generated more lemmas than the procedure could handle efficiently. The *C-reduction* rule of [29] was essentially a sort of lemmaizing, but less powerful and easier to control. Due to these early difficulties, lemmaizing in subgoal-reduction strategies did not receive much attention for some years. More recently, lemmaizing and *caching* [25] in Horn logic were reintroduced successfully in the framework of Prolog technology theorem proving [5, 19]. Caching comprises *success caching* and *failure caching*. The former is conceptually very close to lemmaizing: solutions are stored in a *cache* for fast retrieval, rather than being added as lemmas. The latter adds the capability of using the information that a goal failed before to avoid trying to solve it again. Related techniques, called *memoing* or *tabling*, have been explored independently in logic programming [36]. The experimental work has been followed by the theoretical analysis of [26], which shows that lemmaizing and caching reduce from exponential to linear the amount of duplication in the search spaces of model elimination for problems in propositional Horn logic.

Our intent in this paper is to show that lemmaizing and caching are meta-level inferences that may apply to different types of strategies, including strategies that are not based on subgoal reduction. For this purpose, we consider *semantic resolution* strategies. The reason for this choice is that, among resolution strategies, semantic strategies are those that provide a general notion of "goal", by partitioning the database in a consistent set of "axioms" and a *set of support*

of "goals". We define meta-rules for lemmaizing in semantic-resolution strategies and we give inference rules that implement them. Lemmaizing in model elimination then becomes a special case. This generalization of lemmaizing is significant in at least two ways:

1. Semantic strategies require that all their inferences are *supported*, i.e. have a premise in the set of support. We observe that lemmaizing consists in generating lemmas from the complement of the set of support (e.g., from the axioms in model elimination), that is, lemmas are unsupported inferences. Indeed, lemmaizing in model elimination can be construed as adding some forward-reasoning capability to an otherwise purely backward-reasoning method.

   Semantic-resolution strategies may do forward or backward reasoning depending on how the set of support is defined. If supported inferences are forward inferences, lemmaizing adds backward reasoning to a forward-reasoning strategy; if supported inferences are backward inferences, lemmaizing adds forward reasoning to a backward-reasoning strategy. Therefore, our treatment makes lemmaizing a general technique for combining forward and backward reasoning in semantic resolution.

2. We point out that lemmaizing is a *meta-level rule*. A derivation is made of inference steps, each justified by an inference rule. Lemmaizing derives a lemma based on a fragment of the current derivation. Therefore, it is an inference at the meta-level with respect to the basic inferences.

In the second part of the paper, we describe how contraction inference rules can be incorporated into semantic-resolution strategies. Furthermore, we define a generalized notion of *purity deletion*, and show how it provides additional power in reducing redundancy in these strategies. Roughly speaking, purity deletion is similar to failure caching, although in a forward-reasoning setting. We continue by formalizing success and failure caching in subgoal-reduction strategies as inference rules. We follow the description of these techniques given in [5]. This also includes inference rules for *depth-dependent* caching in model-elimination strategies with iterative deepening, and a discussion of caching and contraction in subgoal-reduction strategies.

In summary, one can have a semantic-resolution strategy that features both contraction and lemmaizing, which are two strengths of contraction-based and subgoal-reduction strategies respectively. Furthermore, contraction-based strategies, unlike subgoal-reduction strategies, are equipped with tools, such as contraction and indexing, to deal with a database of generated and kept clauses. Therefore, while lemmaizing in semantic resolution will certainly need to be restricted, contraction-based strategies might be less sensitive than subgoal-reduction strategies to the risk of generating too many lemmas.

The rest of the paper is organized in the following way. In Section 2 we give a brief summary of semantic resolution and how it plays a role in terms of forward and backward reasoning methods. Section 3 contains the theoretical treatment of lemmaizing as meta-level inference rules for semantic resolution. In Section 4 we define concrete inference rules for lemmaizing in strategies with set of support. In Section 5 we show how to incorporate contraction rules in semantic strategies with lemmaizing, and we present purity deletion. Section 6 explains the relation of our work

on lemmaizing with lemmaizing in model elimination. Sections 7 and 8 cover the formalization of caching and caching with iterative deepening, respectively. A discussion section concludes the paper. The contents of the first five sections of this paper appeared in preliminary form in [9].

2 Semantic resolution strategies

In this section we recall some background material and terminology on *semantic* or *supported* strategies [10]. In *semantic resolution* [30], resolution is controlled by an interpretation I. A given set of clauses S is partitioned into the subset T of all clauses in S that are satisfied by I and its complement $S-T$. Resolution is restricted in such a way that the consistent subset T is not expanded. Only resolution steps with at most one premise from T are allowed: a clause in either T or $S-T$, called *nucleus*, resolves with one or more clauses in $S-T$, called *electrons*, until a resolvent that is false in I, and therefore belongs to $S-T$, is generated. Semantic resolution may also assume an ordering on predicate symbols, and then require that the literal resolved upon in an electron contains the greatest predicate symbol in the electron.

Hyperresolution [27] is semantic resolution where the interpretation I is defined based on sign: in *positive* hyperresolution, I contains all the negative literals, T contains the non-positive clauses, $S-T$ contains the positive clauses, and the electrons are positive clauses (from $S-T$) that resolve with all the negative literals in the nucleus (from T) to generate a positive hyperresolvent. Dually, in *negative* hyperresolution, I contains all the positive literals, T contains the non-negative clauses, $S-T$ contains the negative clauses, and the electrons are negative clauses (from $S-T$) that resolve with all the positive literals in the nucleus (from T) to generate a negative hyperresolvent. Hyperresolution is more restrictive than generic semantic resolution, because resolution steps where both nucleus and electron are in $S-T$ may not happen (e.g., two negative clauses do not resolve).

In *resolution with set of support* [37], a *set of support* (SOS) is a subset of S such that $S-SOS$ is consistent. Only resolution steps with at most one premise from $S-SOS$ are allowed and all generated clauses are added to SOS. To keep the notation uniform, we use $T=S-SOS$ for the consistent subset and $SOS=S-T$ for its complement in all strategies. Resolution with set of support fits in the paradigm of semantic resolution, under the interpretation that the clauses in T are true, the clauses in SOS and all their descendants are false. *Positive resolution* [27] and *negative resolution* are sometimes considered supported strategies where SOS contains the positive or negative clauses, respectively. However, they are not semantic strategies in the proper sense, because they do not partition the clauses based on an interpretation, with the provision that the consistent set is not expanded.

The original idea of *set-of-support strategy* in [37] was that the axioms of a problem usually form a consistent set and that a strategy should not expand such a set, but rather work on the goals. In this interpretation, T contains the axioms, SOS contains the goal clauses (the clauses obtained from the transformation into clausal form of the negation of the target theorem) and the effect of working with a set of support is that most of the work done by the strategy is done on the goals, yielding *backward-reasoning* strategies. The general definitions of semantic

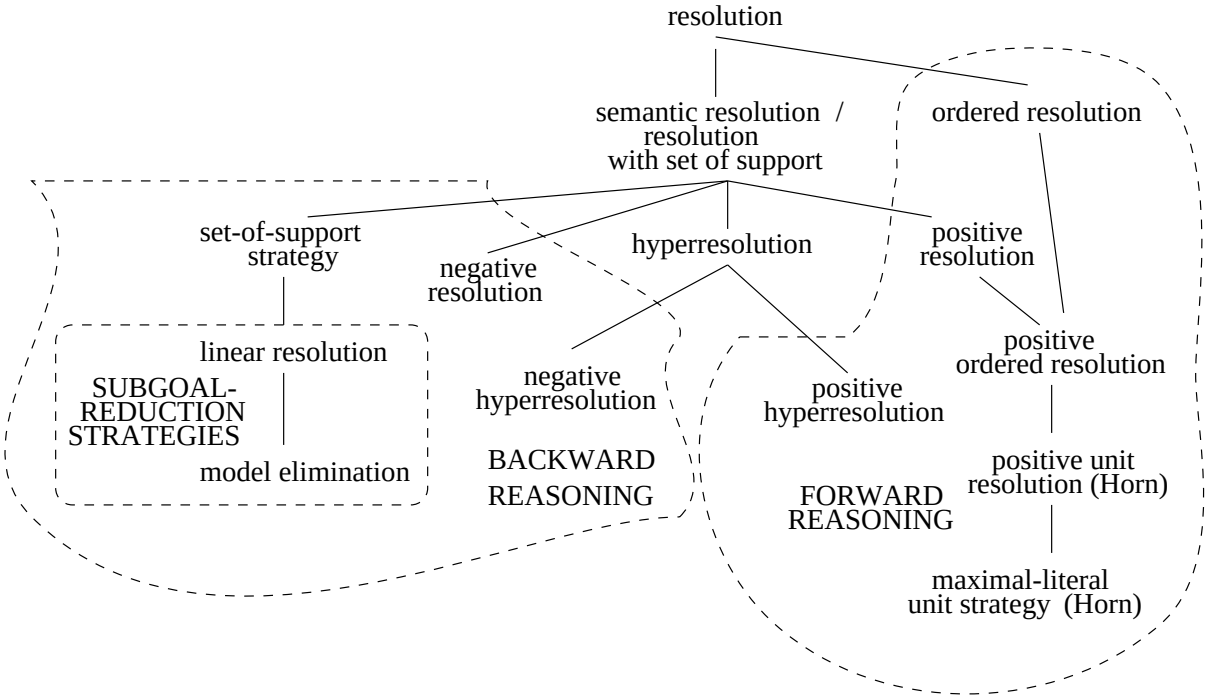

Figure 1: Resolution strategies

resolution and resolution with set of support, however, imply neither backward reasoning nor forward reasoning. For instance, if the axioms are non-negative clauses and the goals are negative clauses, the positive strategies are forward-reasoning strategies and the negative strategies are backward-reasoning strategies compatible with the set-of-support strategy. This is the case in Horn logic. In general, the partition of S into T and SOS based on the distinction between axioms and goals may not agree with the partition based on sign (e.g., the goals may not be negative clauses), so that hyperresolution and the set-of-support strategy are not always compatible.

Linear resolution (see [10] for the many relevant references) can be regarded as a linear refinement of resolution with set of support. Given a set of clauses $S = T \cup \{C_0\}$ with a selected *top clause* C_0, the strategy builds a linear derivation, where at step i clause C_{i+1} is generated by resolving the *center clause* C_i with a *side clause*, either a clause in T (an *input clause*), or a clause C_j such that $j < i$ (an *ancestor clause*). If S is unsatisfiable and T is consistent (C_0 is the negation of the target), there exists a linear refutation of S with C_0 as the top clause. The center clauses form the set of support, and the only needed resolution steps between clauses in SOS are the resolutions with ancestors. Linear resolution is obviously compatible with the set-of-support strategy, and because it is linear, it makes the backward-reasoning character more pronounced: there is a notion of *current goal*, the most recently generated center clause, and each step consists in reducing the current goal to a subgoal. We call such linear, backward-reasoning strategies *subgoal-reduction strategies*.

Linear resolution, however, requires to keep the ancestors around. *Linear input resolution*, where all side clauses are input clauses, is complete for Horn logic, but not for first-order logic. On the other hand, *model elimination* [20] enjoys the advantage of being a linear input strategy

that is complete for first-order logic. Roughly speaking, ancestor-resolution inferences are made unnecessary by saving the literals resolved upon in the goals as *framed literals*, and allowing the latter to resolve away subgoal literals. It follows that each step involves either the current goal and an input clause (analogous to an input resolution step) or the current goal only. Therefore, subgoal-reduction strategies based on model elimination usually work on a stack of goals, rather than on a database of clauses, and at each step focus exclusively on the current goal, on top of the stack. Since there is no database, orderings, contraction and indexing are not used. The search plan is depth-first search with backtracking, and iterative deepening (DFID) [18] to ensure refutational completeness[1]. Finally, since the axiom set of a problem is static, these strategies yield fast implementations using the Warren Abstract Machine [35].

3 Generation of lemmas

In this section we present our treatment of lemmaizing. In Sections 3.1 and 3.2 we give meta-rules for lemma generation in the class of semantic strategies. We assume derivations in the form

$$(T_0; SOS_0) \underset{\mathcal{C}}{\vdash} (T_1; SOS_1) \underset{\mathcal{C}}{\vdash} \ldots (T_i; SOS_i) \underset{\mathcal{C}}{\vdash} \ldots,$$

where $\mathcal{C}$ is any strategy in the class, all generated resolvents are added to the SOS component, but the T component is not assumed to be constant, because it may be modified by contraction or lemmaizing. In Section 4 we give inference rules that implement the meta-rules for lemmaizing and in Section 5 we treat the compatibility of contraction with semantic strategies.

3.1 Generating unit lemmas

In Horn logic, if $T \cup \{\neg L\} \models \Box$, then $T \models L\sigma$ for some substitution σ, and $L\sigma$ can be treated as a lemma of T and be added to T. This also holds in first-order logic if $\Box$ is derived from $T \cup \{\neg L\}$ by a linear input derivation with input set T and top clause $\neg L$. Such a linear input derivation can be produced by a semantic strategy with consistent set T and $\neg L$ in SOS. Then, generalizing this idea slightly, if the strategy deduces a clause $C\sigma$ from $\neg L \vee C$ in SOS by using T alone (without using any other clause in SOS), it means that $T \cup \{\neg L \vee C\} \models C\sigma$. Then $L\sigma$ can also be added as a lemma to T.

There is a caveat, however. For this argument to be sound, it is necessary for the C in $\neg L \vee C$ not to take any part in the derivation of $C\sigma$ from $T \cup \{\neg L \vee C\}$. More precisely, the derivation of $C\sigma$ from $T \cup \{\neg L \vee C\}$ does not include any resolution or factoring step with a selected literal[2] in C. This is necessary to make sure that the existence of a derivation of $C\sigma$ from $T \cup \{\neg L \vee C\}$ implies the existence of a derivation of $L\sigma$ from T. If the derivation from $T \cup \{\neg L \vee C\}$ selects literals in C, then the existence of a derivation of $L\sigma$ from T is not ensured, because the steps

[1] In theorem proving, one is usually interested in a refutation. If the subgoal-reduction tree contains solutions, DFID is guaranteed to find one and halt. In logic programming, on the other hand, one is interested in all solutions. DFID will reach all solutions eventually, but may still fail to terminate if the tree is infinite, deepening forever looking for more solutions.

[2] A selected literal is a literal resolved upon in a resolution step or unified in a factoring step.

selecting literals in C may not be reproducible in a derivation from T. The following definitions will capture this requirement.

Definition 3.1 *Let C be a clause and C' be a binary resolvent or a factor of C. The relation $A \mapsto B$ holds if A is a literal in C different from the selected literal(s) in generating C', B is a literal in C' and $B = A\sigma$, where σ is the most general unifier of the inference generating C'.*

The relation $A \mapsto B$ captures the inheritance of literals that are not selected. By using the transitive closure $\mapsto^*$ of $\mapsto$, we can represent inheritance of literals through a sequence of steps:

Definition 3.2 *Given a resolution derivation $S \vdash^* S'$, where S and S' are sets of clauses, a clause $C' \in S'$ is a* strict descendant *of a clause $C \in S$, if for every literal $A' \in C'$ there is a literal $A \in C$, such that $A \mapsto^* A'$.*

Definition 3.3 *Let S be a set of clauses. $C\sigma$ is* linearly derived from $\neg L \vee C$ by using S *if there is a linear resolution derivation with input set of clauses S, top clause $\neg L \vee C$ and last center clause $C\sigma$. We denote such a derivation by $\neg L \vee C \,|\!\sim\!\!\rightarrow_S C\sigma$.*
If the derivation is a linear input derivation (i.e., all side clauses come from S) and $C\sigma$ is a strict descendant of $\neg L \vee C$, we say that $C\sigma$ is strictly linearly derived from $\neg L \vee C$ by using S *and we write $\neg L \vee C \,|\!\sim\!\!\rightarrow^h_S C\sigma$.*

Coming back to the SOS strategy, given sets SOS and T, $\neg L \vee C \,|\!\sim\!\!\rightarrow^h_T C\sigma$ indicates that $C\sigma$ is derived from $\neg L \vee C$ and T, and that in the derivation, no literals in C and no clauses in SOS are involved in any of the inference steps. We have now all the elements to write the first meta-rule for lemma generation:

Definition 3.4 Unit Lemmaizing*: if $\neg L \vee C \,|\!\sim\!\!\rightarrow^h_T C\sigma$, then add lemma $L\sigma$ to T.*

In order to prove the soundness of Unit Lemmaizing, we generalize Definition 3.3:

Definition 3.5 *Let S be a set of clauses. Clause $C\sigma$ is* linearly derived from $\neg L_1 \vee \cdots \vee \neg L_k \vee C$ by using S*, written $\neg L_1 \vee \cdots \vee \neg L_k \vee C \,|\!\sim\!\!\rightarrow_S C\sigma$, if there exist substitutions $\sigma_1, \cdots \sigma_k$ such that $\forall i$, $1 \leq i \leq k$, $(\neg L_i \vee \cdots \vee \neg L_k \vee C)\sigma_{i-1} \,|\!\sim\!\!\rightarrow_S (\neg L_{i+1} \vee \cdots \vee \neg L_k \vee C)\sigma_i$, where $\sigma_0 = \varepsilon$ (the empty substitution) and $\sigma_k = \sigma$.*

The notion of strictly linear derivation is generalized in the same way with $|\!\sim\!\!\rightarrow^h_S$ at the place of $|\!\sim\!\!\rightarrow_S$. Then we prove a more general result:

Lemma 3.1 *If $\neg L_1 \vee \cdots \vee \neg L_k \vee C \,|\!\sim\!\!\rightarrow^h_T C\sigma$, then $T \models (L_1 \wedge \cdots \wedge L_k)\sigma$.*

Proof: we prove the lemma by induction on the length n of the linear derivation.
Basis: if $n = 1$, then $k = 1$ and there is a unit clause $L' \in T$ such that $L_1\sigma = L'\sigma$. Then we have $T \models L_1\sigma$.

Induction hypothesis: the lemma holds for length n.
Induction step: for $n+1$, let $L' \vee Q_1 \vee \cdots \vee Q_m \in T$ be the side clause used in the first step of the derivation. Without loss of generality, we assume that the first step resolves upon L' and $\neg L_1$ with mgu ρ (i.e., $L'\rho = L_1\rho$). Then the derivation can be refined into:

$$\neg L_1 \vee \cdots \vee \neg L_k \vee C \vdash (Q_1 \vee \cdots \vee Q_m \vee \neg L_2 \vee \cdots \vee \neg L_k \vee C)\rho \overset{h}{\underset{T}{\hookrightarrow}} C\sigma$$

where the first step determines the mgu ρ and the following n steps determine a substitution λ such that $\sigma = \rho\lambda$. By the induction hypothesis, $T \models (L_2 \wedge \cdots \wedge L_k \wedge \neg Q_1 \wedge \cdots \wedge \neg Q_m)\lambda$. Then $T \models (L_2 \wedge \cdots \wedge L_k)\sigma$, since $\lambda \trianglelefteq \sigma$. This takes care of the literals $L_2, \cdots, L_k$. For L_1, we observe that $T \models L' \vee Q_1 \vee \cdots \vee Q_m$ and $T \models \neg Q_i\lambda$ for all i, $1 \leq i \leq m$. It follows that $T \models L'\lambda$, thus $T \models L'\sigma$, because $\lambda \trianglelefteq \sigma$, and $T \models L_1\sigma$. Finally we have $T \models (L_1 \wedge \cdots \wedge L_k)\sigma$. □

The soundness of Unit Lemmaizing follows as a corollary:

Theorem 3.1 *If $\neg L \vee C \,|\!\!\sim^h_T C\sigma$, then $T \models L\sigma$.*

3.2 Generating non-unit lemmas

The derivation $\neg L \vee C \,|\!\!\sim^h_T C\sigma$ in the condition for Unit Lemmaizing satisfies the restrictions that all side clauses of the (linear) derivation come from T and that the literals in C are not selected in the derivation. In Horn logic, since linear input resolution is complete and factoring is not necessary, Unit Lemmaizing is the only form of lemmaizing. In first-order logic, one may have a derivation $\neg L \vee C \,|\!\!\sim_{T \cup SOS} C\sigma$, in which members of the set SOS are also used and $C\sigma$ is not necessarily a strict descendant of C. This condition leads to a more general meta-rule for lemmaizing, that may generate also non-unit lemmas. The general form of a lemma will be $(L \vee F)\sigma$, or $(\neg F \supset L)\sigma$, where $\neg F\sigma$ is the "premise" for $L\sigma$ to hold. Operationally, F contains those subgoals of $\neg L$ that are resolved in the derivation $\neg L \vee C \,|\!\!\sim_{T \cup SOS} C\sigma$ by using SOS or C, but cannot be resolved by using T only. They are formally defined as follows:

Definition 3.6 *Given a derivation $\neg L_1 \vee \cdots \vee \neg L_k \vee C \,|\!\!\sim_{T \cup SOS} C\sigma$, for all i, $1 \leq i \leq k$, the* residue *of $\neg L_i$ in T, denoted by $R_T(\neg L_i)$, is defined as follows.*

- *If $\neg L_i$ is removed by resolution, and $D = L' \vee Q_1 \vee \ldots \vee Q_m$ is the clause that resolves with $\neg L_1 \vee \cdots \vee \neg L_k \vee C$ upon $\neg L_i$ and L', then*

$$R_T(\neg L_i) = \begin{cases} \neg L_i & \text{if } D \in SOS, \\ false & \text{if } D \in T \text{ and } m = 0, \\ R_T(Q_1) \vee \ldots \vee R_T(Q_m) & \text{if } D \in T \text{ and } m \geq 1. \end{cases}$$

- *If $\neg L_i$ is removed by factoring with a literal M, then*

$$R_T(\neg L_i) = \begin{cases} \neg L_i & \text{if } M \text{ is a literal in } C, \\ R_T(\neg L_j) & \text{if } M \text{ is } \neg L_j \text{ for } 1 \leq j \neq i \leq k. \end{cases}$$

Example 3.1 *Assume that* $\neg L \vee C \mid\!\!\rightsquigarrow_{T \cup SOS} C\sigma$ *is made of the following steps:*

1. $\neg L \vee C$ *resolves with* $L \vee P$ *generating* $P \vee C$,
2. $P \vee C$ *resolves with* $\neg P \vee Q \vee R$, *generating* $Q \vee R \vee C$,
3. $Q \vee R \vee C$ *resolves with* $\neg Q$, *generating* $R \vee C$,
4. $R \vee C$ *resolves with* $\neg R$, *generating* C.

We now analyze the residue $R_T(\neg L)$, *according to different situations. If* $L \vee P \in SOS$, *then* $R_T(\neg L) = \neg L$. *If* $L \vee P \in T$, *then* $R_T(\neg L) = R_T(P)$. *In the latter case, if* $\neg P \vee Q \vee R \in SOS$, *then* $R_T(\neg L) = R_T(P) = P$. *On the other hand, if* $\neg P \vee Q \vee R \in T$, *then* $R_T(\neg L) = R_T(P) = R_T(Q) \vee R_T(R)$. *Since* $R_T(Q) = Q$ *if* $\neg Q \in SOS$ *and* $R_T(Q) = false$ *if* $\neg Q \in T$, *and the same is true for* R, *the value of* $R_T(\neg L)$ *in the last case can be determined by the different combinations of* $R_T(Q)$ *and* $R_T(R)$.

A meta-rule for generalized lemmaizing can then be formulated:

Definition 3.7 Generalized Lemmaizing*: if* $\neg L \vee C \mid\!\!\rightsquigarrow_{T \cup SOS} C\sigma$, *then add lemma* $(L \vee R_T(\neg L))\sigma$ *to* T.

Unit Lemmaizing is the special case of Generalized Lemmaizing where $R_T(\neg L)$ is $false$, and thus $(L \vee R_T(\neg L))\sigma$ reduces to $L\sigma$. If $R_T(\neg L)$ is $\neg L$, it means that $\neg L$ itself cannot be resolved in T and therefore no lemma should be added. Indeed, in such a case $(L \vee R_T(\neg L))\sigma$ is a tautology.

Similar to Unit Lemmaizing, the soundness of Generalized Lemmaizing can be obtained as a corollary of a more general result:

Lemma 3.2 *If* $\neg L_1 \vee \cdots \vee \neg L_k \vee C \mid\!\!\rightsquigarrow_{T \cup SOS} C\sigma$, *then for all* i, $1 \leq i \leq k$, $T \models (L_i \vee R_T(\neg L_i))\sigma$.

Proof: similar to Lemma 3.1, the proof is done by induction on the length n of the linear derivation. Basis: if n=1, then $k = 1$. If $\neg L_1$ resolves with a unit clause $L' \in T$ ($L'\sigma = L_1\sigma$), $R_T(\neg L_1) = false$ and the lemma reduces to Lemma 3.1. If $\neg L_1$ resolves with a unit clause $L' \in SOS$ or is eliminated by factoring (with a literal in C), then $R_T(\neg L_1) = \neg L_1$ and the lemma $L_1 \vee R_T(\neg L_1)$ is a tautology.
Induction hypothesis: the lemma holds for length n.
Induction step: for $n+1$, we assume without loss of generality that the first selected literal is L_1. If $\neg L_1$ is resolved upon with side clause $L' \vee Q_1 \vee \cdots \vee Q_m$ and mgu ρ such that $L'\rho = L_1\rho$, we have $(Q_1 \vee \cdots \vee Q_m \vee \neg L_2 \vee \cdots \vee \neg L_k \vee C)\rho \mid\!\!\rightsquigarrow_{T \cup SOS} C\sigma$ in n steps. By the induction hypothesis, $T \models (L_i \vee R_T(\neg L_i))\sigma$ for all i, $2 \leq i \leq k$, and $T \models (\neg Q_i \vee R_T(Q_i))\sigma$ for all i, $1 \leq i \leq m$. If $\neg L_1$ is eliminated by factoring with mgu ρ, we have $(\neg L_2 \vee \cdots \vee \neg L_k \vee C)\rho \mid\!\!\rightsquigarrow_{T \cup SOS} C\sigma$ in n steps, and $T \models (L_i \vee R_T(\neg L_i))\sigma$ for all i, $2 \leq i \leq k$ by induction hypothesis. This takes care of the literals $L_2, \cdots, L_k$. For literal L_1 there are three cases:

- If $L' \vee Q_1 \vee \cdots \vee Q_m \in SOS$ or $\neg L_1$ is eliminated by factoring with a literal in C, then $\neg L_1 \in R_T(\neg L_1)$ and $(L_1 \vee R_T(\neg L_1))\sigma$ is a tautology.

- If $\neg L_1$ is eliminated by factoring with $\neg L_j$ for some j, $2 \leq j \leq k$, then $L_1\rho = L_j\rho$, hence $L_1\sigma = L_j\sigma$, because $\rho \preceq \sigma$, and $R_T(\neg L_1) = R_T(\neg L_j)$, so that $T \models (L_1 \vee R_T(\neg L_1))\sigma$ follows from $T \models (L_j \vee R_T(\neg L_j))\sigma$.
- If $L' \vee Q_1 \vee \cdots \vee Q_m \in T$, then $R_T(\neg L_1) = R_T(Q_1) \vee \cdots \vee R_T(Q_m)$. If $T \models (R_T(Q_1) \vee \cdots \vee R_T(Q_m))\sigma$, the conclusion is proved. Otherwise, it must be that $T \not\models R_T(Q_i)\sigma$ for all i, $1 \leq i \leq m$. Since for all i, $1 \leq i \leq m$, $T \models (\neg Q_i \vee R_T(Q_i))\sigma$ holds by the induction hypothesis, $T \not\models R_T(Q_i)\sigma$ implies $T \models \neg Q_i\sigma$, and therefore $T \not\models Q_i\sigma$ for all i. Then $T \models L'\sigma$, since $L' \vee Q_1 \vee \cdots \vee Q_m \in T$. Since $L'\sigma = L_1\sigma$, we have $T \models L_1\sigma$ and $T \models (L_1 \vee R_T(\neg L_1))\sigma$. □

The following can be easily seen from the above proof:

Corollary 3.1 *Given a derivation $\neg L \vee C \mid\!\!\rightsquigarrow_{T \cup SOS} C\sigma$, if literal $\neg L$ can be eliminated only by a resolution step with a side clause from SOS or by factoring, then no non-trivial lemma can be generated.*

The soundness of Generalized Lemmaizing is a direct consequence of Lemma 3.2:

Theorem 3.2 *If $\neg L \vee C \mid\!\!\rightsquigarrow_{T \cup SOS} C\sigma$, then $T \models (L \vee R_T(\neg L))\sigma$.*

4 Inference rules for Generalized Lemmaizing

In this section we assume that the underlying strategy is resolution with set of support and we give a set of inference rules for resolution and factoring that implement our meta-rules for lemmaizing within such a strategy. In the inference rules the expression $[F]_L$, where F is a disjunction, and L is a literal, is used to denote that F is part of $R_T(L)$. (In order to avoid negation in subscripts, we use L at the place of the $\neg L$ used throughout the previous section). In other words, F is a partial list of subgoals of L resolved away by using SOS. Dually, $\neg F$, a conjunction of literals, is a potential list of premises for resolving away L completely using only clauses in T. When F in $[F]_L$ contains the entire residue of L, the lemma $\neg L \vee F$ can be generated, because $T \models \neg L \vee F$ (equivalently, $T \models \neg F \supset \neg L$, or $T \cup \{\neg F\} \models \neg L$). Some of the inference rules generate a resolvent with literals labelled by a subscript L, such as Q_L. These are subgoals produced while resolving away L, and they themselves need to be resolved away before a lemma concerning L can be generated. We call a literal with subscript L an *L-subgoal*.

The inference rules are separated into three categories, one for resolution, one for factoring, and one for lemmatization.

4.1 The resolution rules

Several different resolution rules are needed since, in a set of support strategy, the sets SOS and T play different roles.

In the first rule for resolution, literal L in $L \vee C$ is resolved with a non-unit clause from T, and a lemma involving $L\sigma$ is initiated:

Resolution with lemma initiation

$$\frac{(T \cup \{\neg L' \vee D\}; SOS \cup \{L \vee C\})}{(T \cup \{\neg L' \vee D\}; SOS \cup \{L \vee C, (D_{L\sigma} \vee [false]_{L\sigma} \vee C)\sigma\})} \quad L\sigma = L'\sigma$$

where L and L' are literals, σ is their most general unifier, and C and D are disjunctions of literals. $D_{L\sigma}$ has the same literals as D, except that they are labelled by a subscript $L\sigma$. These are the $L\sigma$-subgoals. The expression $[false]_{L\sigma}$ means that at this stage the residue of $L\sigma$ (the premise of a potential lemma $\neg L\sigma$) is empty. In this inference rule we assume that neither of the clauses involved in the resolution step has any subscripted literals.

If the T-clause is a unit clause, no lemma is initiated, because lemmaizing would produce an instance of the unit clause in T. Thus, plain unit resolution applies:

Plain Unit Resolution

$$\frac{(T \cup \{L'\}; SOS \cup \{\neg L \vee C\})}{(T \cup \{L'\}; SOS \cup \{\neg L \vee C, C\sigma\})} \quad L\sigma = L'\sigma$$

The next rule covers resolution with a clause from SOS:

Plain resolution

$$\frac{(T; SOS \cup \{L' \vee D, \neg L \vee C\})}{(T; SOS \cup \{L' \vee D, \neg L \vee C, (D \vee C)\sigma\})} \quad L\sigma = L'\sigma$$

We assume that neither of the two clauses involved in the resolution step has any subscripted literals. In this case, if one were to produce a residue for L, it would be L itself (by the first case of Definition 3.6), which would result in a lemma that is a tautology (Corollary 3.1). Thus, there is no need to initiate a lemma.

Residue extension

$$\frac{(T; SOS \cup \{\neg L' \vee Q, P_L \vee D_L \vee C \vee [F]_L\})}{(T; SOS \cup \{\neg L' \vee Q, P_L \vee D_L \vee C \vee [F]_L, (Q_{L\sigma} \vee D_{L\sigma} \vee C \vee [F \vee P]_{L\sigma})\sigma\})} \quad P\sigma = L'\sigma$$

In this rule, F is the residue of L being constructed, and $P_L \vee D_L$ is the disjunction of the L-subgoals to be solved. Since P, an L-subgoal, is resolved with a clause in SOS, the remaining literals coming from that clause also become part of the set of L-subgoals, and P has to be added to the residue. We remark that the clause $\neg L' \vee Q$ may also be labelled. For instance, $\neg L' \vee Q$ may have the form $\neg L'_M \vee E_M \vee B \vee [H]_M$. Then, the above rule may generate two resolvents: $(E_{L\sigma} \vee B_{L\sigma} \vee D_{L\sigma} \vee C \vee [F \vee P]_{L\sigma})\sigma$ and $(D_{M\sigma} \vee C_{M\sigma} \vee E_{M\sigma} \vee B \vee [H \vee \neg L']_{M\sigma})\sigma$.

Subgoal elimination

$$\frac{(T \cup \{\neg L' \vee Q\}; SOS \cup \{P_L \vee D_L \vee C \vee [F]_L\})}{(T \cup \{\neg L' \vee Q\}; SOS \cup \{P_L \vee D_L \vee C \vee [F]_L, (Q_{L\sigma} \vee D_{L\sigma} \vee C \vee [F]_{L\sigma})\sigma\})} \quad P\sigma = L'\sigma$$

This rule is similar to *residue extension* except that the resolved literal P is not added to the residue, because the clause which resolves P away is from T.

4.2 The factoring rules

Similar to the resolution rules, the factoring rules need to consider the behaviour of the L-subgoals and residues.

Residue extension factoring

$$\frac{(T; SOS \cup \{P_L \vee D_L \vee [F]_L \vee C \vee P'\})}{(T; SOS \cup \{P_L \vee D_L \vee [F]_L \vee C \vee P', (D_{L\sigma} \vee [F \vee P]_{L\sigma} \vee C \vee P')\sigma\})} \quad P\sigma = P'\sigma$$

This rule says that if an L-subgoal is eliminated by factoring with a "normal" SOS-literal, then it needs to be considered as part of the residue of L.

Subgoal deletion factoring

$$\frac{(T; SOS \cup \{P_L \vee D_L \vee P'_L \vee C\})}{(T; SOS \cup \{P_L \vee D_L \vee P'_L \vee C, (D \vee P'_{L\sigma} \vee C)\sigma\})} \quad P\sigma = P'\sigma$$

This rule says that, when factoring between two L-subgoals, one of them can be eliminated. This rule corresponds to the fifth case in Definition 3.6.

Plain factoring

$$\frac{(T; SOS \cup \{P \vee D_L \vee P' \vee C\})}{(T; SOS \cup \{P \vee D_L \vee P' \vee C, (D_L \vee P' \vee C)\sigma\})} \quad P\sigma = P'\sigma$$

For the last two rules, the premises may contain framed literals. They are not shown, because the inferences do not change framed literals.

4.3 The lemma generation rule

Lemmaizing

$$\frac{(T; SOS \cup \{[F]_L \vee C\})}{(T \cup \{\neg L \vee F\}; SOS \cup \{C\})} \quad C \textit{ does not contain any } L-\textit{subgoals}$$

In the rule for lemmaizing, all the subgoals of the literal L have been solved. Therefore F is $R_T(L)$ and $\neg L \vee R_T(L)$ is turned into a lemma.

Example 4.1 *If $T = \{P \vee R, \neg R\}$ and $SOS = \{\neg P \vee \neg Q\}$, the first resolvent is $\neg Q \vee R_{\neg P} \vee [false]_{\neg P}$. The new $(\neg P)$-subgoal in the resolvent resolves with $\neg R$ of T and derives $\neg Q \vee [false]_{\neg P}$. By the Lemmaizing rule, the last resolvent becomes $\neg Q$ and a lemma P can be added to T. Note that since set-of-support forbids resolution among members of T, the same lemma cannot be obtained from T directly.*

Example 4.2 *If T contains $\neg P \vee \neg Q$ and SOS contains $P \vee \neg Q$, then $\neg Q_P \vee [false]_P \vee \neg Q$ is inferred by Resolution with lemma initiation. If factoring is applied, the factor $[false \vee \neg Q]_P \vee \neg Q$ is generated. Since $\neg Q$ is the residue of P, the lemma $\neg P \vee \neg Q$ is derived. In this case the lemma is already in T so that it is not added. If T contains Q, and subgoal elimination instead of*

factoring is applied to Q and $\neg Q_P \vee [false]_P \vee \neg Q$, we get $[false]_P \vee \neg Q$. Lemmaizing applied to $[false]_P \vee \neg Q$ adds the unit lemma $\neg P$ to T, and leaves $\neg Q$ in SOS, which will then generate the empty clause with Q in T. Thus, different lemmas may be generated depending on different selections of inferences.

5 Eliminating redundancy in contraction-based strategies

Forward-reasoning resolution strategies often adopt contraction inference rules such as *clausal subsumption* and *clausal simplification* to reduce the database of clauses. Since space explosion is usually the critical factor deciding whether a successful derivation is possible, the more power contraction exhibits, the more effective the proof method is. In this section we discuss two results. First we present schemes of inference rules to incorporate contraction in semantic strategies, including strategies with lemmaizing. Then we present a notion of *purity* with which one can utilize unresolvable literals to detect and delete redundant clauses.

For the first contribution, the combination of the set-of-support strategies and contraction strategies seems to have been implemented in some provers including OTTER, but it is rarely studied in the theorem-proving literature. Similarly, the notion of purity is known since the Davis-Putnam procedure [12] for propositional logic, and may be part of the folk literature for first-order logic. In addition to a formal treatment of purity in first-order logic, we point out an analogy between purity deletion and "failure caching" in Prolog technology theorem proving.

5.1 Incorporating contraction in semantic strategies with lemmaizing

Informally speaking, contraction for semantic strategies is similar to that for plain forward-reasoning strategies, except that some additional care must be taken to guarantee that contraction steps preserve the property of semantic strategies that the set T is consistent. The following example shows that this is not obvious:

Example 5.1 *Assume a semantic strategy featuring clausal simplification and a contraction-first search plan. Let T and SOS be $T = \{\neg P, P \vee Q\}$ and $SOS = \{\neg Q\}$. According to the contraction-first search plan, the strategy looks for contraction steps first, and it contracts $P \vee Q$ in T to P, by applying clausal simplification to $P \vee Q$ and $\neg Q$. It follows that T and SOS become $T = \{\neg P, P\}$ and $SOS = \{\neg Q\}$. The set T has become inconsistent and no refutation can be found by a semantic strategy, since resolution between clauses in T is not allowed.*

Intuitively, this example shows that a careless application of contraction in a semantic strategy may "move" the inconsistency of $T \cup SOS$ into T. Since a semantic strategy assumes that T is consistent, it is not able to detect an inconsistency in T. Therefore, if contraction "moves" the inconsistency in T, it "hides" it to the strategy, making it incomplete. It follows that preserving the consistency of T is a necessary condition for preserving the completeness of the strategy.

Instead of giving an inference rule for each contraction rule in the context of a semantic strategy, we give general schemes of inference rules. In these schemes, we assume that "contraction"

is any sound contraction inference rule that has been proved to preserve the completeness of forward-reasoning resolution strategies. Proper clausal subsumption and clausal simplification are two such rules for first-order theorem proving. Treatments of the completeness of forward-reasoning (ordered) resolution strategies with contraction may be found in the literature (e.g., [6, 15]). In this way, we can focus on the compatibility of contraction with the semantic character of the strategy. We assume that I is the interpretation controlling the semantic strategy. Then the basic idea is that for contraction to be compatible with the semantic strategy, a clause generated by contraction should be added to T only if the clause is true in I:

1. **Contraction of SOS**

$$\frac{(T; SOS \cup \{C\})}{(T; SOS \cup \{D\})} \quad C \textit{ is contracted to } D$$

   If a clause in SOS is contracted, the resulting clause is also false in I and therefore belongs to SOS.

2. **Contraction of T by T**

$$\frac{(T \cup \{C\}; SOS)}{(T \cup \{D\}; SOS)} \quad C \textit{ is contracted to } D \textit{ by clauses in } T$$

   If a clause in T is contracted by clauses in T, the resulting clause is also true in I and therefore belongs to T.

3. **Contraction of T by SOS**

$$\frac{(T \cup \{C\}; SOS)}{(T; SOS \cup \{D\})} \quad C \textit{ is contracted to } D \textit{ by clauses in } SOS \textit{ and } I \not\models D$$

$$\frac{(T \cup \{C\}; SOS)}{(T \cup \{D\}; SOS)} \quad C \textit{ is contracted to } D \textit{ by clauses in } SOS \textit{ and } I \models D$$

   Unlike in the previous schemes, if a clause in T is contracted by clauses in SOS, one needs to resort to the definition of the interpretation I guiding the semantic strategy in order to decide the affiliation of the new clause. Otherwise, incompleteness such as shown in Example 5.1 may occur. In resolution with set of support, all clauses descending from SOS clauses are regarded as false clauses and belong to SOS. Thus, the above two rules can be combined into the following simpler scheme:

$$\frac{(T \cup \{C\}; SOS)}{(T; SOS \cup \{D\})} \quad C \textit{ is contracted to } D \textit{ by clauses in } SOS$$

In case of deletion rules such as subsumption, C is simply deleted and no clause D is generated.

Example 5.2 *If the strategy is resolution with set of support and clausal simplification is applied to the sets* $T = \{\neg P, P \vee Q\}$ *and* $SOS = \{\neg Q\}$ *of Example 5.1, according to Scheme 3, we get* $T = \{\neg P\}$ *and* $SOS = \{\neg Q, P\}$. *A resolution step between* $\neg P \in T$ *and* $P \in SOS$ *completes the proof. In the specific case of clausal simplification, the choice of adding the new clause to* SOS, *if*

the simplifier is in SOS, is also justified operationally by considering that a clausal simplification step can be viewed as the composition of a resolution step and a subsumption step. Thus, we would first resolve $P \vee Q$ and $\neg Q$, yielding $T = \{\neg P, P \vee Q\}$ and $SOS = \{\neg Q, P\}$, where the resolvent P is naturally added to SOS. Then P would subsume $P \vee Q$ leading to $T = \{\neg P\}$ and $SOS = \{\neg Q, P\}$.

Example 5.3 *If the strategy is positive hyperresolution, the set of clauses of Example 5.1 is partitioned into $T = \{\neg P, \neg Q\}$ and $SOS = \{P \vee Q\}$. The clausal simplification step transforms them into $T = \{\neg P, \neg Q\}$ and $SOS = \{P\}$, where P belongs to SOS, because it is positive. In general, for positive hyperresolution, if a (positive) SOS clause is contracted, the resulting clause is also positive, because contraction does not modify the sign, and therefore it belongs to SOS (see Scheme 1). If a clause in T is simplified by another clause in T, both clauses are non-positive. Since the simplifier in clausal simplification is a unit clause, it must be a negative unit clause, that eliminates a positive literal. Thus the resulting clause is also non-positive and remains in T (see Scheme 2). Finally, if a clause in T is simplified by a clause in SOS, the clause in T is non-positive and the simplifier in SOS is a unit positive clause that deletes a negative literal. The resulting clause may be either positive or non-positive and will be added to either SOS or T accordingly. For instance, if $T = \{\neg P \vee \neg Q\}$ and $SOS = \{P, Q\}$, the clausal simplification of $\neg P \vee \neg Q$ by Q gives $T = \{\neg P\}$ and $SOS = \{P, Q\}$, where $\neg P$ stays in T, even if it is simplified by a clause in SOS, because it is negative.*

The following theorem summarizes the compatibility of contraction with semantic resolution strategies:

Theorem 5.1 *Let I_1 denote a resolution inference system, I_1' a semantic restriction of I_1, and I_2 a set of sound contraction inference rules. If*

1. *I_1' is refutationally complete and*
2. *$I_1 \cup I_2$ is refutationally complete,*

then $I_1' \cup I_2$, where the contraction rules in I_2 are applied according to the above schemes, is also refutationally complete.

Proof: let $(T'; SOS')$ be derived from $(T; SOS)$ by a contraction rule in I_2 applied following the above schemes. We need to show that if $T \cup SOS$ is inconsistent and T is consistent, then $T' \cup SOS'$ is inconsistent and T' is consistent. Intuitively, the second part of this thesis (if T is consistent, then T' is consistent) guarantees precisely that contraction does not "move" the inconsistency of $T \cup SOS$ into T, thereby hiding it to the semantic strategy. The first part ($T' \cup SOS'$ is inconsistent) follows from the hypothesis that $I_1 \cup I_2$ is refutationally complete. For the second part (T' is consistent), we observe that if Scheme 1 is applied, T' is equal to T and thus is consistent. If Scheme 2 is applied, a clause C in T is contracted to D by a contraction rule in I_2 applied to premises in T. By the soundness of the rules in I_2, we have $T \models D$. This, together with the hypothesis that T is consistent, implies that $T' = T - \{C\} \cup \{D\}$ is consistent.

In case of Scheme 3, let I be the interpretation used by the strategy and such that $I \models T$. In this scheme, either T' is equal to $T - \{C\}$, or $T' = T - \{C\} \cup \{D\}$. Since $I \models T - \{C\}$ and $I \models D$, T' is consistent. □

For instance, I_2 can contain clausal simplification and clausal subsumption. If I_1 is plain resolution, I_1' can be any semantic restriction. If I_1 is *ordered resolution* as defined in [15], I_1' can be ordered resolution with set of support.

The above schemes and theorem remain valid for resolution with set of support and lemmaizing with lemmas generated according to the rules of Section 4.1. One only needs to take care that whenever a contraction inference rule eliminates a subscripted literal in an SOS clause by applying another SOS clause, the residue is updated. We give an inference rule for clausal simplification as an example:

Clausal simplification of SOS by SOS

$$\frac{(T; SOS \cup \{Q, \neg Q'_L \vee C \vee [F]_L\})}{(T; SOS \cup \{Q, C \vee [F \vee \neg Q']_L\})} \quad Q\sigma = Q' \ for\ some\ \sigma$$

where Q is a unit clause which may be either positive or negative. It is presented here as positive just for convenience. If Q' is not subscripted, the inference rule works in the same way but no residue is added. Notice that the difference between this clausal simplification rule and the subgoal elimination rule of Section 4.1 is that clausal simplification replaces a clause by another one, whereas subgoal elimination adds a new clause.

5.2 Purity deletion

Our treatment of lemmaizing in semantic resolution has shown that also forward-reasoning strategies may employ this technique. Thus, lemmaizing in forward-reasoning strategies correspond to lemmaizing or success caching in subgoal-reduction strategies. On the other hand, the notion of *failure caching* has been used quite effectively in subgoal-reduction strategies, but not at all in forward reasoning. Failure caching says that if a goal literal fails, then it can be used to fail any similar goals in the future. Since there is usually no notion of a goal in a forward-reasoning strategy, it is little wonder that failure caching has not been used in this context.

A goal literal (in a subgoal-reduction strategy) fails if it does not unify with any literal of opposite sign in the given set of clauses. One can in fact adopt this idea to get the opposite effect of lemmaizing. To be more precise, if a literal cannot be resolved away, then obviously it cannot play a role in any derivation of refutation. Therefore, any clause which contains such a literal can be deleted. In fact, this idea already existed in the Davis-Putnam procedure [12], in which such a literal was called a *pure* literal. We adopt the name and generalize the notion by the following inductive definition:

Definition 5.1 *Let S be a set of clauses and A be a literal occurring in S.*

1. *If for all clause $C \in S$ there is no literal $B \in C$, such that $A\sigma = \neg B\sigma$ for some substitution σ, then A is* pure *in S.*

2. *If for all the clauses $C \in S$ that contains a B such that $A\sigma = \neg B\sigma$ for some σ, C contains an instance of a pure literal, then A is* pure *in S.*

Condition 1 (*basic purity*) is the basis of the definition, and Condition 2 is the inductive case, that represents a sort of *transitive closure* of purity. Its logical justification is that a clause that contains a pure literal is not necessary for the refutation, and therefore a literal that can resolve only with unnecessary clauses is also unnecessary, or pure.

Proposition 5.1 *If a literal A is pure, then any instance of A is also pure.*

Proof: If an instance, $A\tau$, of A unifies with some literal B of opposite sign, i.e. $A\tau\rho = \neg B\rho$ for some ρ, then A and B are also unifiable. Therefore by definition, $A\tau$ is pure. □

Our inference rule states that any clause which contains a pure literal can be deleted:

Purity deletion

$$\frac{S \cup \{C\}}{S}\ \exists A \in C,\ A \textit{ is pure}$$

If clauses that contain pure literals are deleted, more literals may become pure. The inductive case of the definition of purity captures the propagation of basic purity caused by the application of the Purity Deletion rule: if all the clauses that A may resolve with contain a pure literal, all such clauses will be deleted by the Purity Deletion rule and therefore A will become pure according to basic purity. The inductive part of the definition "anticipates" this propagation effect. Therefore, in order to show that clauses containing pure literals can be deleted while preserving refutational completeness, it is sufficient to consider the basis of the definition of purity.

In propositional logic, if S is an unsatisfiable set of clauses and A is pure in S, then $S' = S - \{C|A \in C\}$ is also unsatisfiable. This is because if S' were satisfiable, there would be a Herbrand model I of S'. Since neither A nor $\neg A$ appears in S', neither of them needs to be in I. Then $I \cup \{A\}$ would be a model of S, contradicting the fact that S is unsatisfiable. The same reasoning does not apply in this form to a set of first-order clauses for the following reason: if A is a ground first-order literal, the fact that neither A nor $\neg A$ appears in S' does not imply that neither of them is in I, because A may be in the Herbrand base of S' even if it does not occur in S', and therefore either A or $\neg A$ may be in I. A mapping of ground first-order atoms into propositional variables, however, is sufficient to extend the reasoning to sets of ground first-order clauses:

Lemma 5.1 *Let S be a finite set of ground first-order clauses and $A(\bar{t})$ be a pure literal in S. If S is unsatisfiable, then $S' = S - \{C|A(\bar{t}) \in C\}$ is unsatisfiable.*

Proof: let $B_0(\bar{s}_0), B_1(\bar{s}_1), \ldots B_n(\bar{s}_n)$ be an enumeration without repetitions of all the atoms that occur in S. $A(\bar{t})$ is $B_k(\bar{s}_k)$ for some k, $1 \leq k \leq n$. For each atom $B_i(\bar{s}_i)$ in this list, we create a distinct propositional variable L_i, so that there is a bijection between the set $\{L_0, L_1, \ldots L_n\}$ and the set $\{B_0(\bar{s}_0), B_1(\bar{s}_1), \ldots B_n(\bar{s}_n)\}$. Let PS be the set of propositional clauses that is obtained from S by replacing each occurrence of $B_i(\bar{s}_i)$ by L_i for all i, $1 \leq i \leq n$. Since S is unsatisfiable,

PS is also unsatisfiable. Since $B_k(\bar{s}_k)$ is pure in S, L_k is pure in PS. Let PS' be the set $PS' = PS - \{C | L_k \in C\}$. Since PS is unsatisfiable, PS' is unsatisfiable by the propositional argument given above. Since $S' \subseteq S$, all atoms that occur in S' also occur in S and thus in the enumeration $B_0(\bar{s}_0), B_1(\bar{s}_1), \ldots B_n(\bar{s}_n)$. Let PS'' be the set of propositional clauses that is obtained from S' by replacing each occurrence of $B_i(\bar{s}_i)$ by L_i for all i, $1 \leq i \leq n$. The two sets PS' and PS'' are equal, and therefore PS'' is unsatisfiable. It follows that S' is unsatisfiable. □

By applying the Herbrand theorem, the lemma can then be lifted to a set of general first-order clauses:

Theorem 5.2 *Let S be a finite set of first-order clauses and $A(\bar{t})$ a pure literal in S. If S is unsatisfiable, then $S' = S - \{C | A(\bar{t}) \in C\}$ is unsatisfiable.*

Proof: since S is unsatisfiable, by the Herbrand theorem, there exists a finite unsatisfiable set S_0 of ground instances of clauses in S. There are two mutually exclusive and exhaustive cases:

- For all $C' \in S_0$ there is a $C \in S'$ such that $C' = C\sigma$ for some substitution σ. In other words, S_0 is a finite unsatisfiable set of ground instances of clauses in S'. It follows that S' is unsatisfiable by the Herbrand theorem.
- There exists a clause $C' \in S_0$ such that for no $C \in S'$ it is $C' = C\sigma$. Thus, it must be $C' = C\sigma$ for some $C \in S - S'$, that is, for some C that contains $A(\bar{t})$. Then $A(\bar{t})\sigma$ occurs in S_0. We prove by way of contradiction that $A(\bar{t})\sigma$ is pure in S_0. If $A(\bar{t})\sigma$ is not pure in S_0, then $\neg A(\bar{t})\sigma$ occurs in S_0. It follows that there is a clause D in S, that contains a literal $\neg A(\bar{s})$ of the same sign of $\neg A(\bar{t})\sigma$, and such that $\neg A(\bar{t})\sigma$ is an instance of $\neg A(\bar{s})$, i.e. $A(\bar{t})\sigma = A(\bar{s})\rho$. Therefore $A(\bar{t})$ and $\neg A(\bar{s})$ in S are unifiable and have opposite signs, contradicting the hypothesis that $A(\bar{t})$ is pure in S. This proves that $A(\bar{t})\sigma$ is pure in S_0. Let S_0' be the set $S_0' = S_0 - \{C' | C' = C\sigma,\ C \in S,\ A(\bar{t}) \in C\}$. S_0' is unsatisfiable by Lemma 5.1 because S_0 is unsatisfiable. Since S_0' is an unsatisfiable set of ground instances of clauses in S', S' is unsatisfiable by the Herbrand theorem. □

In summary, a pure literal in the forward-reasoning context has some similarity with a failed goal of subgoal-reduction strategies, since both notions are based on the impossibility of unifying the literal. Indeed, instances of pure literals are pure, like instances of failed goals also fail. In this sense, purity deletion echoes failure caching in forward-reasoning strategies.

6 Model elimination

In the previous sections we have presented our approach to integrate lemmaizing and contraction in semantic strategies. In this and the following sections, we focus more closely on subgoal-reduction strategies. First, we explain how lemmaizing in model elimination is covered by our meta-rules of Section 3. Because of the importance of model elimination as the foundation of subgoal-reduction strategies, we also report the inference rules of model elimination itself, thereby completing the

survey of strategies of Section 2. This is done in the present section. The following two sections will be dedicated to *caching* and *depth-dependent caching*, that is caching in strategies with iterative deepening. Caching techniques are usually presented in operational terms. We shall give inference rules that capture formally the working of the caching mechanisms. Thus, these inference rules will fill the gap between the logical justification of caching, e.g., our meta-rules of Section 3, and its implementations.

Model elimination was introduced independently of resolution in [20]. In addition to its original formulation, model elimination may also be presented as a refinement of linear resolution (e.g., [21]) and as a tableaux-based method (e.g., [19, 34]). For the purposes of this paper, it is appropriate to adopt the view of model elimination as a refinement of linear resolution.

In model elimination, clauses are treated as ordered lists of literals, called *chains*. The derivation proceeds like a linear resolution derivation, with the provision that literals in the center clauses are resolved in a pre-defined order, e.g., from left to right. A key feature is that at each step the literal resolved upon in the center clause parent is saved as a *framed literal* in the resolvent:

ME-extension

$$\frac{(T \cup \{L' \vee D\}; \neg L \vee C)}{(T \cup \{L' \vee D\}; (D \vee [\neg L] \vee C)\sigma)}\ L\sigma = L'\sigma$$

Saving the literals resolved upon allows the strategy to know exactly when resolution with an ancestor is needed: when the leftmost literal in the current center clause unifies with a framed literal of opposite sign in the same clause. Resolution with ancestor is not applied as such, but it is replaced by a specific inference rule:

ME-reduction

$$\frac{(T; \neg L \vee D \vee [L'] \vee C)}{(T; (D \vee [L'] \vee C)\sigma)}\ L\sigma = L'\sigma$$

The set of inference rules is completed by adding **ME-contraction**, that removes any leftmost framed literal in the current chain, and **ordered factoring**: if the leftmost non-framed literal in the current chain unifies with a non-framed literal of the same sign, a factor may be generated by removing the leftmost literal and applying the unifier. In the context of tableau-based methods, factoring is also called **merging** (e.g., [34]). In addition to making model elimination a linear input strategy, ME-reduction keeps the center clauses short, because ME-reduction simply removes the leftmost literal, whereas ancestor resolution replaces it by the literals other than the one resolved upon in the ancestor. Model elimination recognizes that in a linear derivation it is unnecessary to reintroduce these literals from an ancestor, because they are already being solved in the derivation. This appears in examples where resolution needs factoring to eliminate the extra literals, whereas ME-reduction makes it unnecessary for model elimination.

The idea of putting frames around literals in the system of inference rules of Section 4 was inspired by model elimination, although the purposes are different. The primary purpose of framed literals in model elimination is ME-reduction, whereas in our system the purpose of framed literals is lemmatization. The resulting effects are dual: in model elimination framed literals are *ancestor literals*, while in our system they are *residue literals*, hence *subgoal literals*. In model elimination framed literals are literals that were resolved away by ME-extension (corresponding

to input resolution, hence resolution with clauses in T), while there is no need to frame a literal resolved by ME-reduction (corresponding to ancestor resolution, hence resolution with clauses in SOS), because it is not an ancestor of other literals (and it would be immediately deleted by ME-contraction). Dually, in our system framed literals are literals that were resolved away by clauses in SOS, while there is no need to frame a literal resolved away by clauses in T, because it is not part of the residue.

6.1 Lemmaizing in model elimination

Lemmaizing in model elimination is based on the observation that when an ME-contraction step deletes a framed literal $[\neg L]$ that is the leftmost in the current chain, it means that all the subgoals of $\neg L$ have been solved and therefore $\neg L$ itself has been solved. Having solved goal $\neg L$ means $T \cup \{\neg L\} \models \Box$, that is, $T \models L$. Thus, L can be added as a lemma to T and may be used by ME-extension to solve other goal literals. A unit lemma, however, may be generated only in the absence of ME-reduction steps. If $[A_1], \ldots, [A_n]$ were used in ME-reduction steps to eliminate literals on the left of L, it means that $\neg A_1, \ldots, \neg A_n$ are subgoals of L that cannot be solved by T alone:

ME-contraction with Lemmaizing

$$\frac{(T; [\neg L] \vee C_1 \vee [A_1] \vee \ldots \vee C_n \vee [A_n] \vee C_{n+1})}{(T \cup \{L \vee \neg A_1 \vee \ldots \vee \neg A_n\}; C)} \quad \begin{array}{l}\text{if } [A_1], \ldots, [A_n] \text{ were used by ME-reduction} \\ \text{to solve subgoals of } \neg L.\end{array}$$

In [20] the literals $[A_1], \ldots, [A_n]$ were identified by a *scoping* mechanism: each framed literal has a scope; whenever a framed literal is used in an ME-reduction step, its scope is incremented; when ME-contraction with Lemmaizing is applied, a calculation of scopes and relative positions of framed literals determines which framed literals need to be in the lemma. Intuitively, the scope of a framed literal represents how far to the left that literal has been used: $[A_1], \ldots, [A_n]$ are the framed literals whose scope includes $\neg L$. In our treatment, $\neg A_1 \vee \ldots \vee \neg A_n$ is the *residue* of $\neg L$.

Similar to factoring in semantic resolution, the combination of lemmaizing and merging in model elimination causes the generation of non-unit lemmas:

Example 6.1 *We reconsider the clauses of Example 4.2 in the context of model elimination. If T contains $\neg P \vee \neg Q$ and the current chain is $P \vee \neg Q$, ME-extension derives $\neg Q \vee [P] \vee \neg Q$. If merging is applied, we obtain $[P] \vee \neg Q$, hence $\neg Q$. If lemmaizing is applied when $[P]$ is removed, the generated lemma is $\neg P \vee \neg Q$, because the subgoal P has been solved only under the condition that its subgoal $\neg Q$ will be solved. If the strategy does not feature merging, no lemma may be generated at this point. If T contains Q, ME-extension applied to Q and $\neg Q \vee [P] \vee \neg Q$ gives $[P] \vee \neg Q$, hence $\neg Q$, and the unit lemma $\neg P$ may be generated. At the next step the empty clause is reached.*

Lemmaizing in the context of tableaux-based methods is treated in [19] and [34], where it is called *regressive merging*, because its effect on the tableaux appears dual to that of merging.

7 Inference rules for caching

If a strategy generates all the lemmas that can be generated, the set T (and therefore the search space) may be expanded in such a way that the original characteristic of the strategy of not expanding T is defeated, and the performance of the strategy may suffer rather than improve. A simple criterion to limit the number of lemmas generated is to generate only unit lemmas. Since all lemmas in Horn logic are unit lemmas, it may be feasible to generate and save all of them. For the purpose of efficient implementation, however, strategies for Horn logic often employ *caching* rather than lemmaizing. Caching has been used by many strategies in both theorem proving and logic programming (e.g., [5, 25, 36]). In this and the following sections, first we summarize the basic ideas in caching, and then we formalize the caching mechanisms following [5] as a set of inference rules.

7.1 Caching

Caching has the same logical justification as lemmaizing: if $\neg L \vee C \mid\!\!\sim\!\!\rightarrow^{h}_{T} C\sigma$, then $L\sigma$ is a logical consequence of T. Lemmaizing consists in adding $L\sigma$ to T and let the strategy use lemmas as premises of inferences like any other clause in T. Caching differs from lemmaizing in that it stores the solved goal literal $\neg L$ and its solution σ in a fast data structure called *cache*, rather than adding $L\sigma$ to T as lemma. Furthermore, the information stored in the cache is not used to provide premises to the regular inference mechanism. Rather, whenever the current goal contains an instance of $\neg L$, this instance is solved by an operation of cache look-up that retrieves the solutions of $\neg L$. This is called *success caching*. In *failure caching*, the information that a goal literal was tried and failed is also stored in the cache, and used to fail its instances. In both cases, the idea is to solve or fail goals by reusing the results of previous deductions, without performing other inference steps. In terms of search, caching allows to solve or fail goals based on the portion of search space of the problem visited so far, without expanding it further.

The motivation for caching is that cache retrieval is faster than inferences. However, replacing inferences by cache retrieval has two consequences. First, because cache retrieval depends on the current goal, caching can be used by strategies that have a notion of current goal, that is, subgoal-reduction strategies. On the other hand, lemmaizing can be used by a more general class of strategies, because it makes the lemmas available to whatever inference system the strategy has. Second, both lemmaizing and caching are conceived as enhancements of strategies that are already complete. However, because cache retrieval replaces, whenever applicable, the regular inference mechanism, it is necessary to cache *all the solutions* in order to retain completeness. This requirement does not apply to lemmaizing, where there is no restriction to the application of the inference mechanism, and one is interested in saving only those lemmas that appear useful.

7.2 Inference rules for caching in Horn logic

We first discuss the relatively simple case of caching for Horn logic.

We use a component *Cache* to represent the cache, and the notation $A \hookrightarrow A\rho_1, \ldots, A\rho_r$ to

indicate a cache entry storing solutions $A\rho_1, \ldots, A\rho_r$ of the goal literal A. The symbol $\hookrightarrow$ has no logical meaning, it simply says that matching the pattern A in the cache gives access to the solutions $A\rho_1, \ldots, A\rho_r$. To simplify the notation, we use $A \hookrightarrow A\rho$ when a specific solution $A\rho$ is being inserted or retrieved. If A failed, the corresponding cache entry is $A \hookrightarrow \emptyset$. We emphasize that the cache contains goals and goal solutions, not lemmas. In terms of the Unit Lemmaizing rule, if $\neg L \vee C \mid\!\!\sim^h_T C\sigma$, then $L\sigma$ is the lemma, and $\neg L \hookrightarrow \neg L\sigma$ is the corresponding cache entry. To avoid negative signs, we use A and A' rather than $\neg L$ as goal literals. By expressing the rules for success caching in terms of the meta-rule for unit lemmaizing, we make our formulation independent from any specific subgoal-reduction strategy.

We consider first the insertion of data in the cache. An entry is inserted in the cache when a goal literal succeeds or fails. A goal literal fails if it does not unify with any literal of opposite sign in the set of axioms (*Basic Failure*) or if all its subgoals fail (*Recursive Failure*):

Insert cache entry

- *Success*

$$\text{If } A \vee G \mid\!\!\sim^h_T G\rho, \text{ then add } A \hookrightarrow A\rho \text{ to } Cache$$

  where G is the disjunction of the remaining goal literals.

- *Basic Failure*

$$\frac{(T; A \vee G; Cache)}{(T; fail; Cache \cup \{A \hookrightarrow \emptyset\})} \;\; \forall C \in T,\ \forall B \in C,\ \neg\exists\sigma \text{ such that } A\sigma = \neg B\sigma$$

  where $A\sigma = \neg B\sigma$ means that A and B unify and have opposite signs.

- *Recursive Failure*

$$\frac{(T; A \vee G; Cache)}{(T; fail; Cache \cup \{A \hookrightarrow \emptyset\})} \;\; \begin{array}{l} \forall Q_1 \vee \ldots \vee Q_m \in T, \text{ such that } \exists\sigma\ A\sigma = \neg Q_1\sigma, \\ (Q_2 \vee \ldots \vee Q_m \vee G)\sigma \text{ fails.} \end{array}$$

The strategy uses cached solutions of a goal to generate solutions of other goals that are generated later. If all the solutions of a solved goal A' are $A'\rho_1, \ldots, A'\rho_r$, then all the solutions of an instance $A = A'\sigma$ are those $A'\rho_j$ that are instances of A. Thus, if all the solutions of A' are in the cache, the solutions of A can be found by cache retrieval, with no need to use T (*success caching*). Since the cache stores solutions, cache retrieval will consist essentially of a *matching* operation: matching A with the solutions of A' in the cache. (In lemmaizing, the lemmas $\neg A'\rho_j$'s would be added to T, and A would be solved by a unit-resolution step with one such lemma.) If A' failed, the information that A' failed is retrieved from the cache and used to fail A (*failure caching*). Indeed, if A had a solution $A\rho$, this solution would also be a solution of A', as $A'\sigma\rho$, contradicting the fact that A' failed:

Retrieve cache entry

- *Success*

$$\frac{(T; A \vee G; Cache \cup \{A' \hookrightarrow A'\rho\})}{(T; G\tau; Cache \cup \{A' \hookrightarrow A'\rho\})} \;\; A = A'\sigma,\ A'\rho = A'\sigma\tau$$

- *Failure*

$$\frac{(T; A \vee G; Cache \cup \{A' \hookrightarrow \emptyset\})}{(T; fail; Cache \cup \{A' \hookrightarrow \emptyset\})} \; A = A'\sigma$$

7.3 Inference rules for caching in first-order logic

Caching may be *inconsistent* with inference rules for first-order logic. In operational terms, this is because caching assumes that the subgoal-reduction process is *context-free*, in the sense that each goal literal is reduced independently of its context. In logical terms, caching assumes that each solution corresponds to a unit lemma. These assumptions hold in Horn logic, where all lemmas are unit lemmas and context-free reduction, such as in ME-extension, is sufficient. In first-order logic, reduction is not context-free (e.g., ME-reduction and factoring/merging are context-sensitive rules), and there are non-unit lemmas (non-empty residues). Caching is inconsistent, because it is not correct to solve A by matching it with a solution $A'\rho$ of A' in the cache, if the residue of A' is not empty. If the residue is not empty, Generalized Lemmaizing deduces $(\neg A' \vee R_T(A'))\rho$, not $\neg A'\rho$. Caching can be made compatible with subgoal-reduction strategies for first-order logic by keeping track of the residue in the cache:

Insert cache entry

$$\text{If } A' \vee G \mid\!\!\sim_{T \cup SOS} G\rho, \text{ then add } A' \hookrightarrow A'\rho :: R_T(A')\rho \text{ to } Cache$$

where the notation :: is simply a separator between the solution and the residue.

Then, a proceeding goal $A \vee G$ such that $A = A'\sigma$ can be reduced to $G\tau$ by using the cache entry $A' \hookrightarrow A'\rho :: R_T(A')\rho$ only if $A'\rho = A\tau$ (the retrieved solution is an instance of A) and $G\tau$ contains $R_T(A')\rho$. This ensures that solving $G\tau$ will solve the subgoals of A' that must be solved for $A'\rho$ to be a solution of A' and thus of A:

Retrieve cache entry

$$\frac{(T; A \vee G; Cache \cup \{A' \hookrightarrow A'\rho :: R_T(A')\rho\})}{(T; G\tau; Cache \cup \{A' \hookrightarrow A'\rho :: R_T(A')\rho\})} \begin{array}{l} A = A'\sigma,\ A'\rho = A'\sigma\tau \\ \forall B \in R_T(A')\rho,\ \exists G_i \in G,\ B = G_i\tau \end{array}$$

In logical terms, resolving the lemma $(\neg A' \vee R_T(A'))\rho$ with the goal $A \vee G$ generates the new goal $R_T(A')\rho \vee G\tau$. If the subgoals in $R_T(A')\rho$ are in $G\tau$ (the condition of the above rule), $R_T(A')\rho \vee G\tau$ is trivially equivalent to $G\tau$.

The reasoning is essentially the same for model elimination. ME-extension of the chain $A \vee G$ by the lemma $(\neg A' \vee R_T(A'))\rho$ gives $R_T(A')\rho \vee G\tau$. If for all literals $B \in R_T(A')\rho$, there is a framed literal $[G_i]$ in G, such that $B = G_i\tau$ and B and G_i have opposite signs, then all the literals in $R_T(A')$ are eliminated by ME-reduction, and $R_T(A')\rho \vee G\tau$ reduces to $G\tau$. Thus, under this condition, a goal $A \vee G$ such that $A = A'\sigma$ can be reduced to $G\tau$ in one caching step by using the entry $A' \hookrightarrow A'\rho :: R_T(A')\rho$. Since in model elimination one works with ancestors, rather than with residues, the condition to apply a cache entry is naturally expressed in terms of ancestors: a goal literal A can be solved by retrieving from the cache a solution of A' that was obtained by using ancestors of A', only if A has the same ancestors [3, 34]. Under this formulation, a cache entry stores the ancestors, e.g. $A' \hookrightarrow A'\rho :: \mathcal{L}\rho$, where $\mathcal{L}\rho$ is the list of ancestors of A' that were

used to generate the solution $A'\rho$. Then, a proceeding goal $A \vee G$ such that $A = A'\sigma$ can be reduced to $G\tau$ by using $A' \hookrightarrow A'\rho :: \mathcal{L}\rho$ if $A'\rho = A\tau$ and for all $B \in \mathcal{L}\rho$, there is a $[G_i]$ in G, such that $B = G_i\tau$. This formulation is equivalent to the previous one, because in model elimination residue literals and ancestor literals differ only in the sign (see Section 6.1).

We conclude this section by observing that the reliance of the caching mechanism on a context-free subgoal-reduction process, makes caching *incomplete* in the presence of context-sensitive pruning rules. In model elimination, one such rule is *Identical ancestor pruning*

$$\frac{(T; A \vee G_1 \vee [A] \vee G_2)}{(T; fail)}$$

where a search path is pruned if a subgoal is identical to an ancestor. Caching is incompatible with identical ancestor pruning [5]. If the search tree for A' was pruned based on a certain context, such as the presence of an ancestor, there is no guarantee that all the solutions of A can be found by matching with the solutions of A'. This is because the context-sensitive condition of the pruning rule may not be satisfied on the corresponding paths in the search tree for A. If the regular search mechanism were used, those paths in the search tree for A would not be pruned, and might yield additional solutions of A that are not solutions of A'. If cache retrieval replaces the regular search mechanism, these additional solutions of A will not be computed. In practice, a theorem proving may feature both a strategy with caching and a strategy with identical ancestor pruning and apply them separately.

7.4 Caching and contraction

In order to work with contraction rules, a strategy needs to keep generated clauses, so that kept clauses may be used to delete other generated clauses that are redundant. For instance, if a strategy does not keep clauses, it cannot use them to subsume other clauses. Pure subgoal-reduction strategies do not keep generated clauses, they work exclusively with the current goal and the input axioms. Of course, the strategy may backtrack to a past goal, but it does not build a database of clauses. Thus, pure subgoal-reduction strategies do not feature contraction. In this sense, the use of contraction is intrinsically connected to forward reasoning: forward reasoning needs contraction to keep the size of the database in check, and forward reasoning makes contraction possible in the first place by generating the database.

Since lemmaizing (or caching) adds some forward reasoning to subgoal-reduction strategies, it also makes some form of contraction applicable. Indeed, a subgoal-reduction strategy with lemmaizing may feature subsumption among lemmas, or *lemma subsumption*. *Cache subsumption* is the subsumption among solutions of a goal in the cache [5, 36]. From a practical point of view, cache subsumption has a definite advantage over lemma subsumption for subgoal-reduction strategies. The reason is that many subgoal-reduction strategies are implemented under the assumption that elements in T will not be deleted. Therefore, adding lemma subsumption in T may be problematic. On the other hand, caching adds goal solutions to the cache, not to T, so that it is possible to augment the inference system with cache subsumption without disturbing the basic working of the strategy.

We consider how the rules given in Section 7.2 need to be modified to take cache subsumption into account. Failure caching obviously remains unchanged. Success caching is modified as follows. For the insertion of solutions, whenever the strategy tries to insert a solution $A'\rho$ of A', it tests whether $A'\rho$ is subsumed by some other solution $A'\theta$ of A' already in the cache. If $A'\rho$ is subsumed, there is no need to add it. If $A'\rho$ is not subsumed, the strategy checks whether $A'\rho$ subsumes any pre-existing solution of A', and finally adds $A'\rho$ to the cache. For the retrieval of solutions, the application of cache subsumption means that only the most general solutions are kept in the cache. Therefore, unification, rather than matching, must be applied in order to generate a solution of a goal literal A from a cache entry $A' \hookrightarrow A'\rho$:

$$\frac{(T; A \vee G; Cache \cup \{A' \hookrightarrow A'\rho\})}{(T; G\tau; Cache \cup \{A' \hookrightarrow A'\rho\})}\ A = A'\sigma,\ A'\rho\tau = A'\sigma\tau$$

We observe that a cache is organized in such a way that all the solutions of A' can be accessed together, so that the subsumption tests of cache subsumption can be executed very rapidly. In other words, the cache itself provides a form of indexing.

In summary, exactly because caching is a form of forward reasoning, it is not compatible with all pruning mechanisms that are typical of subgoal-reduction strategies (e.g. identical ancestor pruning), but it makes contraction rules that are typical of forward-reasoning strategies (e.g. subsumption) available to subgoal-reduction strategies.

8 Inference rules for depth-dependent caching

The rules for caching given in the previous section are *depth-independent*: a goal literal fails regardless of when it is selected during the search process, and cached solutions are used independently of the depth where they were found. In most backtracking-based strategies, exhaustive search is implemented via iterative deepening, in which it is possible to define depth-dependent notions of failure and success, and therefore depth-dependent caching mechanisms. In this section, we assume a subgoal-reduction strategy with a DFID search plan, and we present inference rules for depth-dependent caching in Horn logic. Let k be the depth bound at the current round of iterative deepening and Q be the initial goal. A pair (G, n) denotes a goal with associated depth bound n ($0 \leq n \leq k$): this means that $k - n$ steps were used to reduce Q to G, and n more steps are allowed to solve G as shown in Figure 2.

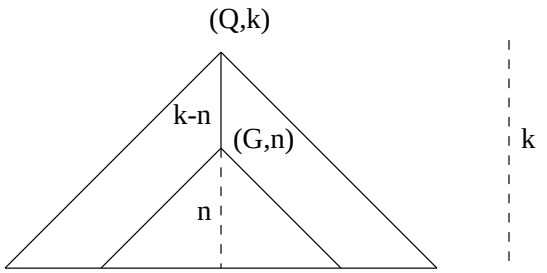

Figure 2: Depth bound associated to a goal

The notation (A, n) has the same meaning for a single goal literal A. The depth bound associated to the current goal is decremented at each inference step:

$$\frac{(T; (G, n); k)}{(T; (G', n-1); k)}$$

If the depth bound of the goal reduces to 0 before a solution is found, the goal, e.g. $(G, 0)$, fails. This means that the allowed depth is not sufficient to find a solution. We call this type of failure *Basic Depth-dependent Failure*, in addition to the Basic Failure and Recursive Failure of Section 7.2. A failure (regardless of its type) will cause either one of two behaviours:

1. backtrack to an ancestor within depth limit k, e.g. to some (G', m), $0 < m \leq k$, for which there are still choice-points open;

2. if there is no such ancestor, i.e. the search space down to depth k has been exhausted, reset the limit to some $k + i$, for $i > 0$, and start the next round of iterative deepening with the query $(Q, k + i)$.

We have now all the elements to write rules for depth-dependent caching. A solution from the cache can be used for goal (A, n) only if it could have been found by search within the depth bound n associated to A. Therefore, solutions need to be stored with an associated depth bound. Let $(A \vee G, n)$, where $n \leq k$, be the given goal. Assume that its literal (A, n) is resolved upon, and after $n - q$ steps, where $0 \leq q \leq n$, all subgoals of (A, n) have been resolved away, so that (A, n) is solved with an answer substitution ρ. This solution is written $(A\rho, n - q)$, to say that it took $n - q$ steps (starting from $(A \vee G, n)$, not from the input goal) to generate the solution $A\rho$, as shown in Figure 3.

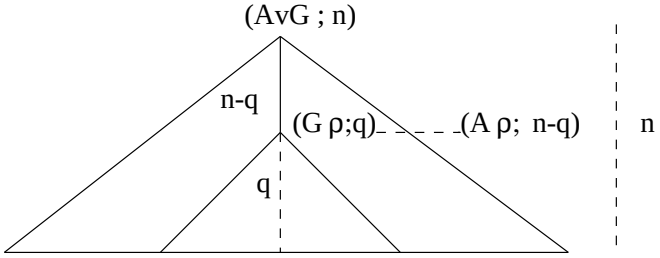

Figure 3: Depth bound associated to a solution

In summary, the depth bound associated to a goal indicates how deep the search procedure can go to solve that goal, whereas the depth bound associated to a solution indicates the depth where that solution was found.

Insert depth-dependent cache entry

- *Success*

$$\text{If } (A \vee G, n) \,|\!\sim\!\!\!\rightarrow^{h}_{T} (G\rho, q), \text{ then add } (A, n) \hookrightarrow (A\rho, n - q) \text{ to } Cache$$

Remark that $n-q \leq n$, that is, the depth bound associated to a solution is bounded by the depth bound associated to the corresponding goal. After solving A, the current goal is $(G\rho, q)$, because we still have to solve $G\rho$ and we have q steps to do it within the current depth bound.

The insertion of entries in the cache upon failure is modified slightly. There is no need to introduce a caching rule for Basic Depth-dependent Failure, because adding to the cache an entry in the form $(A, 0) \hookrightarrow \emptyset$ is redundant. The rule for Basic Failure of Section 7.2 is unchanged, except that given a current goal $(A \vee G; n)$, where A cannot be resolved away, an entry $(A, n) \hookrightarrow \emptyset$, rather than $A \hookrightarrow \emptyset$ is added to Cache. The rule for Recursive Failure is changed in the same way. In addition, the condition for Recursive Failure changes, because the base of the recursion may be provided by any combination of Basic Failure and Basic Depth-dependent Failure.

A goal literal can be failed based on the cache if a more general goal with a higher associated depth bound already failed:

Retrieve depth-dependent cache entry

- *Failure*

$$\frac{(T; (A \vee G, n); k; Cache \cup \{(A', m) \hookrightarrow \emptyset\})}{(T; fail; k; Cache \cup \{(A', m) \hookrightarrow \emptyset\})}\ A = A'\sigma,\ 0 < n \leq m \leq k$$

If it was not possible to find a solution for A' by searching down to depth m, there must be no solution for A within depth $n \leq m$. Indeed, if A had a solution $A\rho$ that can be found within depth n, then A' would have a solution $A'\sigma\rho$ that can be found within depth m, since $n \leq m$, a contradiction. The condition $n \leq m \leq k$ also implies that $k - m \leq k - n$: since (A', m) was selected after $k - m$ steps, and (A, n) is selected after $k - n$ steps, this means that (A', m) was selected *before* (A, n).

On the success side, a goal literal (A, n) can be solved by cache look-up, if a more general goal literal (A', m), with a higher associated depth bound $(n \leq m)$, was already solved, and had solutions that can be found within the depth n allowed for A. More precisely, if a goal literal (A', m) was solved and $(A'\rho_1, p_1), \ldots, (A'\rho_r, p_r)$, where $p_j \leq m$ for all j, $1 \leq j \leq r$, are all its solutions, then all the solutions of a goal literal (A, n), such that $A = A'\sigma$, and $n \leq m$, are those $(A'\rho_j, p_j)$ such that $p_j \leq n$, and $A'\rho_j$ is an instance of A:

- *Success*

$$\frac{(T; (A \vee G, n); k; Cache \cup \{(A', m) \hookrightarrow (A'\rho, p)\})}{(T; (G\tau, n - p); k; Cache \cup \{(A', m) \hookrightarrow (A'\rho, p)\})}\ A = A'\sigma,\ 0 < p \leq n \leq m \leq k,\ A'\rho = A\tau$$

The depth bound associated to the remaining goal $G\tau$ is $n - p$, because a solution of depth p has been used. The cache must contain *all* the solutions of (A', m) to guarantee that *all* the solutions of (A, n) may be found by caching. We remark that when we say "all the solutions" of a goal in the form (A, n), we mean "all the solutions within depth n".

The variants of the caching rules described in the previous section to make caching compatible with first-order strategies and to keep into account cache subsumption, can be extended to depth-dependent caching. Cache subsumption also becomes depth-dependent: given two solutions $(A'\rho_1, p_1)$ and $(A'\rho_2, p_2)$ of (A', m) such that $A'\rho_1$ subsumes $A'\rho_2$, $(A'\rho_2, p_2)$ can be deleted if

$p_1 \leq p_2$, because we do not need a solution of A' that is less general and requires to search deeper in order to be found. On the other hand, if $p_1 > p_2$, then $(A'\rho_2, p_2)$ must be kept, because it yields a solution for all those goals (A, n) with $A = A'\sigma$ such that $p_2 \leq n \leq m$ but $p_1 > n$. $(A'\rho_1, p_1)$ would not yield a solution for such an (A, n), because $p_1 > n$.

An additional variant of caching in the depth-dependent case is *Heuristic Caching* [5], where the depth bound associated to the current goal is decreased by an heuristical value h, $0 < h < p$, rather than by the depth p of the applied solution:

$$\frac{(T; (A \vee G, n); k; Cache \cup \{(A', m) \hookrightarrow (A'\rho, p)\})}{(T; (G\tau, n - h); k; Cache \cup \{(A', m) \hookrightarrow (A'\rho, p)\})}\ A = A'\sigma,\ 0 < h < p \leq n \leq m \leq k,\ A'\rho = A\tau$$

The advantage of this heuristic is that the strategy has more steps available to solve $G\tau$, since $h < p$. Thus, it may find more solutions in the current round of iterative deepening than it would if pure caching (i.e., caching without heuristic) or no caching were applied. The disadvantage is that the portion of search space searched within the current round of iterative deepening will be larger than with pure caching or no caching: it may happen that no additional solutions are found, while more memory and time are consumed.

Caching and iterative deepening are especially well-suited for one another. For instance, depth-dependent failure caching may apply more often than basic failure caching, because there is a notion of depth-dependent failure. Depth-dependent failure is a relative failure: it may be that (A', m) fails while (A', p) for some $p > m$ succeeds. All (A, n) with $A = A'\sigma$ and $n \leq m$, however, can be failed based on the failure of (A', m), thereby saving a significant amount of redundant search. On the other hand, basic failure caching does not apply until A' has failed absolutely. Similarly, basic success caching does not apply until all solutions of A' have been found. Depth-dependent success caching only needs that all solutions within a given depth have been found. Therefore, depth-dependent caching allows to use the cache more eagerly than basic caching.

9 Discussion

Forward-reasoning resolution strategies for first-order logic often suffer from generating too many irrelevant clauses. In order to control the growth of the database of clauses, contraction inference rules are usually employed. Subgoal-reduction strategies, on the other hand, lack the ability of producing useful lemmas which may reduce the search effort. The question of how to combine the best of the two worlds has long been a challenge to the automated deduction community.

In this paper we address this question by working on the idea of lemmaizing. Lemmaizing is a concept used in the logic programming community for saving previous execution results for later use. It has been used effectively both in enhancing Prolog [36] and in Prolog technology theorem proving [5]. We show how to generalize lemmaizing to semantic resolution, formulating the principle of lemmaizing as meta-level inference rules. From this starting point we give the following contributions towards integrating features of forward and backward reasoning:

- We define concrete inference rules that implement the meta-rules for lemmaizing in forward-

reasoning strategies with set of support. This shows that lemmaizing, a native feature of goal-oriented strategies, can be done also in forward reasoning.

- We show how contraction rules can be incorporated in semantic strategies, even with lemmaizing. This means that lemmaizing, a backward-reasoning feature, and contraction, a forward-reasoning feature, can coexist. Furthermore, contraction rules can take advantage of the unit lemmas produced by lemmaizing.

- We present a redundancy deletion method for forward-reasoning strategies which is based on a notion of purity.

- We explain how our meta-rules for lemmaizing cover the lemmaizing of model elimination as a special case.

- We formalize the caching techniques of subgoal-reduction strategies as inference rules justified by the lemmaizing meta-rules, including the depth-dependent caching of strategies with a DFID search plan.

- We observe that lemmaizing and caching, by allowing subgoal-reduction strategies to keep some generated clauses, make available to these strategies forms of contraction, such as subsumption, that are typical of forward-reasoning strategies.

Our approach has a number of advantages. Unlike previous work on lemmatization which is mostly limited to model elimination, it provides a general way of adding lemmas to the complement of the set of support. Therefore, it offers a flexible way of adding some forward-reasoning ability to goal-oriented strategies or, vice versa, some backward-reasoning ability to forward strategies. The fact that our meta-rules for lemmaizing provide a logical foundation for many techniques, including lemmaizing in set-of-support strategies, lemmaizing in model elimination, caching and depth-dependent caching, shows that these meta-rules capture the essence of lemmaizing independently of specific strategies and even classes of strategies. Our work on purity highlights an intuitive correspondence between purity deletion and failure caching. This is complementary to the intuitive correspondence between subsumption and success caching suggested in [33], and therefore reinforces the understanding that while contraction eliminates redundancy in contraction-based strategies, caching eliminates redundancy in subgoal-reduction strategies.

We conclude with a discussion of the relation between this work and logic programming, considering logic programming with Horn clauses and the less known paradigm of logic programming with rewrite rules.

The dichotomy of forward reasoning and subgoal-reduction that we have described for theorem proving is very well-known also in logic programming and deductive databases. Top-down evaluation of Horn clauses is a subgoal-reduction strategy, and some of its strengths (e.g., it is goal-oriented) and weaknesses (e.g., the repetition of subgoals) are essentially the same as those of subgoal-reduction strategies for theorem proving. Symmetrically, bottom-up evaluation of Horn clauses is a forward-reasoning strategy: it is not goal-oriented, but since it proceeds by generating and keeping facts, it can use them to eliminate duplicates, which is a simple form of contraction. Techniques similar to lemmaizing and especially caching, known under the names of *memoing*,

tabling or *OLDT-resolution*, have been developed independently in logic programming with the purpose of preventing repetition of subgoals in top-down evaluation[3]. On the other hand, in deductive databases, the *magic template* or *magic sets* transformation of Horn clauses without function symbols (Datalog programs) has been designed to make bottom-up evaluation sensitive to the query. A survey of these approaches may be found in [36].

While it is well-known that the tabling techniques in logic programming are essentially forms of lemmaizing (at least from a logical point of view that abstracts from the sophistication of actual implementations), we observe that also the magic template can be interpreted in terms of lemmaizing. This is best shown by means of a simple example taken from [36]: the (semi-naïve) bottom-up evaluation of the clauses

(1) $arc(a, b)$.
(2) $arc(c, b)$.
(3) $arc(b, d)$.
(4) $path(X, Y) :- arc(X, Y)$.
(5) $path(X, Z) :- arc(X, Y), path(Y, Z)$.

generates first the facts $\{arc(a, b), arc(c, b), arc(b, d)\}$, then $\{path(a, b), path(c, b), path(b, d)\}$, and finally $\{path(c, d), path(a, d)\}$. The magic transformation driven by the query $? - path(c, X)$ replaces clauses (4) and (5) by

(6) $path(X, Y) :- calls\_to\_path(X, Y), arc(X, Y)$.
(7) $path(X, Z) :- calls\_to\_path(X, Z), arc(X, Y), path(Y, Z)$.
(8) $calls\_to\_path(c, X)$.
(9) $calls\_to\_path(Y, Z) :- calls\_to\_path(X, Z), arc(X, Y)$.

The bottom-up evaluation of the transformed program generates first $\{arc(a, b), arc(c, b), arc(b, d), calls\_to\_path(c, X)\}$, then adds $\{path(c, b), calls\_to\_path(b, Z)\}$ at the first iteration, $\{path(b, d), calls\_to\_path(d, Z)\}$ at the second iteration, and $\{path(c, d)\}$ at the third iteration, after which the fixed-point is reached. The evaluation is partially goal-driven in that the facts $path(a, b)$ and $path(a, d)$, that are irrelevant to the query, are not generated. What is interesting to this discussion, however, is the generation of the facts $calls\_to\_path(b, Z)$ and $calls\_to\_path(d, Z)$. Since $path(b, Z)$ and $path(d, Z)$ are subgoals of $path(c, X)$, the magic transformation has forced bottom-up evaluation to generate subgoals, that is, lemmas from the goal. Of course, it is well-known that while tabling techniques add a bottom-up character to top-down evaluation, magic sets add a top-down character to bottom-up evaluation. We would like to emphasize that both mechanisms are an instance of lemmatization from the logical point of view: lemmas from the axioms in tabling and lemmas from the goal in magic sets. Magic sets are different, because in magic sets lemma generation is induced by transforming the program, rather than by enriching the inference system. For instance, in the above example, the inference system is positive unit resolution (the inference rule which corresponds to bottom-up evaluation) in both executions. However, in the second execution the presence of clauses (8) and (9) introduced by the magic transformation causes the inference system to generate the lemmas $calls\_to\_path(b, Z)$ and $calls\_to\_path(d, Z)$.

We observe that lemmatization appears also in logic programming with *rewrite programs* and

[3]In logic programming the main motivation for avoiding the repetition of subgoals is to improve the termination properties of programs.

Linear Completion. Rewrite programs are made of rewrite rules interpreted as logical equivalences of conjunctions of atoms: facts have the form $A \rightarrow true$, and rules may have the form $A, B_1, \ldots B_n \rightarrow B_1, \ldots B_n$ (meaning A if $B_1, \ldots B_n$), or $A \rightarrow B_1, \ldots B_n$ (meaning A iff $B_1, \ldots B_n$). A query $\exists \bar{x}\ Q_1, \ldots Q_k$ is written as $Q_1, \ldots Q_k \rightarrow answer(\bar{x})$, and if a rule in the form $answer(\bar{x})\sigma \rightarrow true$, called *answer rule*, is deduced, σ is an answer substitution. The Linear Completion of rewrite programs is basically a subgoal-reduction strategy, where at each step a program rule *overlaps* with the current goal to generate a new goal. In [7], Linear Completion also features *Simplification* of the current goal by its ancestor goals, and the *addition of the generated answer rules* to the program. The use of simplification, coupled with the additional expressive power of iff-rules, modifies the behaviour of the programs in several ways, including some loop avoidance and pruning of the search space, that were studied in detail in [7]. Here we emphasize that the addition of the answer rules to the program is another form of lemmaizing. Indeed, Linear Completion treats the answer rules like the input program rules, using them for overlap and simplification of the current goal. Also, both simplification and lemmatization of answers are forward-reasoning features added to a subgoal-reduction mechanism.

In summary, lemmatization is a powerful idea, which covers seemingly disparate mechanisms, such as memoing, magic sets and answer rules, emerged in different approaches to logic programming. In this paper, we have shown that lemmatization can be done in full first-order logic, and for inference systems as general as semantic resolution. In our treatment, lemmaizing may be used to add forward-reasoning to backward-reasoning strategies and backward-reasoning to forward-reasoning strategies, thereby extending to first-order logic and theorem proving the duality that was known in logic programming.

Acknowledgements

We would like to thank Mark Stickel for answering our questions on caching in model elimination.

References

[1] S. Anantharaman and M. P. Bonacina. An application of automated equational reasoning to many-valued logic. In M. Okada and S. Kaplan, editors, *Proceedings of the Second International Workshop on Conditional and Typed Term Rewriting Systems*, number 516 in Lecture Notes in Computer Science, pages 156–161, Montréal, Canada, June 1990. Springer Verlag.

[2] S. Anantharaman and J. Hsiang. Automated proofs of the Moufang identities in alternative rings. *Journal of Automated Reasoning*, 6(1):76–109, 1990.

[3] O. L. Astrachan. *Investigations in theorem proving based on model elimination.* PhD thesis, Dept. of Computer Science, Duke University, 1992.

[4] O. L. Astrachan and D. W. Loveland. METEORs: High performance theorem provers using model elimination. In R. S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Automated Reasoning Series, pages 31–60. Kluwer Academic Publisher, Dordrecht, The Netherlands, 1991.

[5] O. L. Astrachan and M. E. Stickel. Caching and lemmaizing in model elimination theorem provers. In D. Kapur, editor, *Eleventh Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 224–238, Saratoga Springs, New York, U.S.A., June 1992. Springer Verlag. Full version available as Tech. Rep. 513 SRI International, December 1991.

[6] L. Bachmair and H. Ganzinger. On restrictions of ordered paramodulation with simplification. In M. E. Stickel, editor, *Tenth Conf. on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 427–441. Springer Verlag, 1990.

[7] M. P. Bonacina and J. Hsiang. On rewrite programs: semantics and relationship with Prolog. *Journal of Logic Programming*, 14(1 & 2):155–180, October 1992.

[8] M. P. Bonacina and J. Hsiang. Towards a foundation of completion procedures as semidecision procedures. *Theoretical Computer Science*, 146:199–242, July 1995.

[9] M. P. Bonacina and J. Hsiang. On semantic resolution with lemmaizing and contraction. In N. Foo and R. Goebel, editors, *Fourth Pacific Rim Int. Conf. on Artificial Intelligence*, volume 1114 of *Lecture Notes in Artificial Intelligence*, pages 372–386, Cairns, Australia, August 1996. Springer Verlag.

[10] C. L. Chang and R. C. Lee. *Symbolic logic and mechanical theorem proving*. Academic Press, New York, U.S.A., 1973.

[11] J. D. Christian. Fast Knuth-Bendix completion: summary. In N. Dershowitz, editor, *Third Conf. on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 551–555, Chapel Hill, North Carolina, U.S.A., April 1989. Springer Verlag.

[12] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[13] N. Dershowitz. Canonical sets of Horn clauses. In J. Leach Albert, B. Monien, and M. Rodríguez Artalejo, editors, *Eighteenth Int. Conf. on Automata, Languages and Programming*, volume 510 of *Lecture Notes in Computer Science*, pages 267–278, Madrid, Spain, 1991. Springer Verlag.

[14] S. Fleisig, D. Loveland, A. Smiley, and D. Yarmush. An implementation of the model elimination proof procedure. *Journal of the ACM*, 21:124–139, 1974.

[15] J. Hsiang and M. Rusinowitch. Proving refutational completeness of theorem proving strategies: the transfinite semantic tree method. *Journal of the ACM*, 38:559–587, 1991.

[16] D. Kapur and H. Zhang. RRL: a rewrite rule laboratory. In E. Lusk and R. Overbeek, editors, *Nineth Conf. on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 768–770, Argonne, Illinois, U.S.A., May 1988. Springer Verlag.

[17] D. Kapur and H. Zhang. A case study of the completion procedure: proving ring commutativity problems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 360–394. The MIT Press, Cambridge, Massachusetts, 1991.

[18] R. E. Korf. Depth-first iterative deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[19] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: a high performance theorem prover. *Journal of Automated Reasoning*, 8(2):183–212, 1992.

[20] D. W. Loveland. A simplified format for the model elimination procedure. *Journal of the ACM*, 16(3):349–363, 1969.

[21] D. W. Loveland. A unifying view of some linear Herbrand procedures. *Journal of the ACM*, 19(2):366–384, 1972.

[22] W. McCune. Experiments with discrimination tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[23] W. McCune. Otter 3.0 reference manual and guide. Technical Report 94/6, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

[24] W. McCune. Solution of the Robbins problem. Pre-print of the Mathematics and Computer Science Division, Argonne National Laboratory, 1997.

[25] D. A. Plaisted. Non-Horn clause logic programming without contrapositives. *Journal of Automated Reasoning*, 4(3):287–325, 1988.

[26] D. A. Plaisted. The search efficiency of theorem proving strategies. In A. Bundy, editor, *Twelfth Conf. on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 57–71. Springer Verlag, 1994. Full version available as Tech. Rep. of the Max Planck Institut für Informatik, MPI-I-94-233.

[27] J. A. Robinson. Automatic deduction with hyper-resolution. *International Journal of Computer Mathematics*, 1:227–234, 1965.

[28] M. Rusinowitch. Theorem-proving with resolution and superposition. *Journal of Symbolic Computation*, 11(1 & 2):21–50, 1991.

[29] R. E. Shostak. Refutation graphs. *Artificial Intelligence*, 7:51–64, 1976.

[30] J. R. Slagle. Automatic theorem proving with renamable and semantic resolution. *Journal of the ACM*, 14(4):687–697, 1967.

[31] M. E. Stickel. A Prolog technology theorem prover: implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4:353–380, 1988.

[32] M. E. Stickel. The path-indexing method for indexing terms. Technical Report 473, SRI International, 1989.

[33] M. E. Stickel. PTTP and linked inference. In R. S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Automated Reasoning Series, pages 283–296. Kluwer Academic Publisher, Dordrecht, The Netherlands, 1991.

[34] K. Wallace and G. Wrightson. Regressive merging in model elimination tableau-based theorem provers. *Journal of the IGPL*, 3(6):921–937, 1995.

[35] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, 1983.

[36] D. S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):94–111, 1992.

[37] L. Wos, D. Carson, and G. Robinson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the ACM*, 12:536–541, 1965.