CS-1994-19

Proof Procedures for Logic Programming

Donald W. Loveland and Gopalan Nadathur

Department of Computer Science

Duke University

Durham, North Carolina 27708–0129

May 1994

Proof Procedures for Logic Programming†

Donald W. Loveland
Gopalan Nadathur
Computer Science Department
Duke University
Durham, NC 27706, USA

Abstract

Proof procedures are an essential part of logic applied to artificial intelligence tasks, and form the basis for logic programming languages. As such, many of the chapters throughout this handbook utilize, or study, proof procedures. The study of proof procedures that are useful in artificial intelligence would require a large book so we focus on proof procedures that relate to logic programming. We begin with the resolution procedures that influenced the definition of SLD-resolution, the procedure upon which Prolog is built. Starting with the general resolution procedure we move through linear resolution to a very restricted linear resolution, SL-resolution, which actually is not a resolution restriction, but a variant using an augmented logical form. (SL-resolution actually is a derivative of the Model Elimination procedure, which was developed independently of resolution.) We then consider logic programming itself, reviewing SLD-resolution and then describing a general criterion for identifying inference systems that can serve as bases for logic programming. This criterion requires, in essence, that logical symbols exhibit a duality between a declarative and a search-related meaning and we present the notion of a uniform proof as a means for formalizing such a duality. We describe some first-order systems that extend the Horn clause logic while satisfying this criterion. The usefulness of these extended systems from the perspective of programming and the construction of proof procedures for them is then examined. The third section discusses two proposed logic programming languages that extend Prolog in a manner that overcomes some of its limitations. Specifically, these proposals involve the addition of hypothetical implication and positive disjunctions. These extensions are viewed from the perspective of constructing uniform proofs.

1 Building the Framework: The Resolution Procedure

A proof procedure is an algorithm (technically, a semi-decision procedure) which identifies a formula as valid (or unsatisfiable) when appropriate, and may not terminate when the formula is

† This paper is to appear in the *Handbook of Logic in AI and Logic Programming*, D. Gabbay, C. Hogger and A. Robinson (eds.), Oxford University Press. Loveland's work on this paper was partially supported by NSF Grants CCR-89-00383 and CCR-91-16203 and Nadathur's work was similarly supported by NSF Grants CCR-89-05825 and CCR-92-08465.

invalid (satisfiable). Since a proof procedure concerns a logic the procedure takes a special form, superimposing a search strategy on an inference calculus. We will consider a certain collection of proof procedures in the light of an inference calculus format that abstracts the concept of logic programming. This formulation allows us to look beyond SLD-resolution, the proof procedure that underlies Prolog, to generalizations and extensions that retain an essence of logic programming structure.

The inference structure used in the formulation of the logic programming concept and first realization, Prolog, evolved from the work done in the subdiscipline called Automated Theorem Proving. While many proof procedures have been developed within this subdiscipline, some of which appear in Volume 1 of this handbook, we will present a narrow selection, namely the proof procedures which are clearly ancestors of the first proof procedure associated with logic programming, SLD-resolution. Extensive treatment of proof procedures for automated theorem proving appear in Bibel [Bib82], Chang & Lee [CL73] and Loveland [Lov78].

1.1 The Resolution Procedure

Although the consideration of proof procedures for automated theorem proving began about 1958 we begin our overview with the introduction of the Resolution proof procedure by Robinson in 1965. We then review the linear resolution procedures, Model Elimination and SL-resolution procedures. Our exclusion of other proof procedures from consideration here is due to our focus, not because other procedures are less important historically or for general use within automated or semi-automated theorem process.

After a review of the general resolution proof procedure, we consider the linear refinement for resolution and then further restrict the procedure format to linear input resolution. Here we are no longer capable of treating full first-order logic, but have forced ourselves to address a smaller domain, in essence the renameable Horn clause formulas. By leaving the resolution format, indeed leaving traditional formula representation, we see there exists a linear input procedure for all of first-order logic. This is the Model Elimination(ME) procedure, of which a modification known as the SL-resolution procedure was the direct inspiration for the SLD-resolution procedure which provided the inference engine for the logic programming language Prolog. The ME - SL-Resolution linear input format can be translated into a very strict resolution restriction, linear but not an input restriction, as we will observe.

The resolution procedure invented by Alan Robinson and published in 1965 (see [Rob65]) is studied in depth in the chapter on resolution-based deduction in Volume 1 of this handbook, and so, quickly reviewed here. The resolution procedure is a refutation procedure, which means we establish unsatisfiability rather than validity. Of course, no loss of generality occurs because a formula is valid if and only if (iff) its negation is unsatisfiable. The formula to be tested for unsatisfiability is "converted to conjunctive normal form with no existential quantifiers present". By this we mean that given formula F to be tested for unsatisfiability, we associate a logically equivalent formula F' which is in conjunctive normal form (a conjunction of clauses, each clause a disjunction of atomic formulas or negated atomic formulas, called *literals*) with no existential quantifiers. F' may be written without any quantifiers, since we regard F, and therefore F', as a closed formula, so the universal quantifiers are implicitly assumed to be present preceding each clause. (Thus we are free to rename variables of each clause so that no variable occurs in more than one clause.) The formula F' is derivable from F by standard logical equivalences plus the use of Skolem functions to replace quantifiers. We shall say that F' is in *Skolem conjunctive form* and that F' is the *Skolem conjunctive form of* F. Precise algorithms for conversion are given in many textbooks, including Chang & Lee [CL73] and Loveland [Lov78].

The formula $\sim P(x, f(x)) \vee Q(x, g(x), z)$ is the Skolem conjunctive form of

$$\forall x[P(x, f(x) \supset \exists y \forall z Q(x, y, z)],$$

where $g(x)$ is a Skolem function introduced for the y that occurs in the intermediate formula

$$\forall x[\sim P(x, f(x)) \vee \exists y \forall z Q(x, y, z)].$$

(Recall that an existentially quantified variable is replaced by a new function letter followed by arguments consisting of all universally quantified variables where the universal quantifier contains the existential quantifier within its scope.) The formula

$$\forall z \exists x \forall y (A(y, y) \vee [B(y, z) \wedge C(x, y)])$$

has

$$(A(y, y) \vee B(y, z)) \wedge (A(w, w) \vee C(a, w))$$

as its Skolem conjunctive form formula, where a is a Skolem function having 0 arguments. Note that in this example immediate Skolemization, before applying the distributive law and then importing the quantifiers, would have introduced a Skolem function $f(z)$ instead of the Skolem constant a used above. For pragmatic reasons regarding the cost of search in automated proof procedures it is best to seek Skolem functions with the fewest number of arguments. We have adopted the traditional logic notation of capital letters for predicate letters, and non-capitals for variables, function and constant letters. Variables are represented by end of the alphabet letters.

For those accustomed to logic programming notation, based on clauses as implications, it is a natural question to ask: why use conjunctive normal form? The resolution procedure (and hence its restrictions) does not distinguish between positive literals (atomic formula, or atoms) and negative literals (negated atoms). Also, the general resolution procedure (and many variants) requires symmetric calling access to all literals of a clause. Thus the symmetry of OR as the connective in the clause is very suitable. Of course, we could present the entire set of procedures using implicative form but there is no insight gained and the presentation would differ from the traditional presentations for no essential reason. Therefore, we will invoke a notation change in the middle of the paper, but this reflects common sense and historical precedent, so we forsake uniform notation.

Resolution procedures are based on the resolution inference rule

$$\frac{C_1 \vee a \quad \sim a \vee C_2}{C_1 \vee C_2}$$

where $C_1 \vee a$ and $\sim a \vee C_2$ are two known clauses, with C_1 and C_2 representing arbitrary disjuncts of literals. The literals a and $\sim a$ are called the *resolving literals*. The derived clause is called the *resolvent*. The similarity of the inference rule to Gentzen's cut rule is immediately clear and the rule can be seen as a generalization of *modus ponens*. The resolvent is true in any model for the two given clauses, so the inference rule preserves validity.

The resolution inference rule just given is the propositional rule, also called the *ground* resolution rule for a reason given later. We postpone discussion of the very similar first-order inference rule to delay some complications.

Because the Skolem conjunctive format is so uniform in style, it is convenient to simplify notation when using it. We drop the OR symbol in clauses and instead simply concatenate literals. Clauses themselves are either written one per line or separated by commas. When the non-logical symbols are all single letters it is also convenient to drop the parentheses associated with predicate letters and the commas between arguments. Thus the Skolem conjunctive form formula $(P(a) \vee P(b)) \wedge (\sim P(a)) \wedge (\sim P(b))$ is shortened to $PaPb, \sim Pa, \sim Pb$ by use of the simplifying notation.

Using the just-introduced shorthand notation for a formula in this form, often called a *clause set*, we present a refutation of a clause set. We use the terms *input clause* to designate a clause from the *input* clause set given by the user to distinguish such clauses from derived clauses of any type.

1.	$PaPb$	input clause
2.	$Pa{\sim}Pb$	input clause
3.	${\sim}PaPb$	input clause
4.	${\sim}Pa{\sim}Pb$	input clause
5.	Pa	resolvent of clauses 1, 2
6.	${\sim}Pa$	resolvent of clauses 3, 4
7.	contradiction	Pa and ${\sim}Pa$ cannot both hold

Note that in line 5 the two identical literals are merged.

Since each resolvent is true in any model satisfying the two parent clauses, a satisfiable clause set could not yield two contradictory unit clauses. Thus Pa, $\sim Pa$ signals an unsatisfiable input clause set. The resolution inference rule applied to Pa and $\sim Pa$ yields the empty clause, denoted $\Box$. This is customarily entered instead of the word "contradiction", as we do hereafter.

We have noted that the ground resolution inference system is *sound*, i.e. will not deduce $\Box$ if the original clause set is satisfiable. We now note that this inference system is complete, i.e. for every propositional unsatisfiable clause set there is a resolution refutation. A proof appears in Chang & Lee [CL73] or Loveland [Lov78]. A proof by induction on the number of "excess" literal occurrences in the clause set (the number of literal occurrences minus the number of clauses in the set) is not difficult and the reader may enjoy trying the proof himself.

Above, we have given a proof of unsatisfiability, or a refutation. There is no indication there as to how such a proof might be found. Descriptions of search algorithms that employ the resolution inference rule comprise proof procedures for resolution. Perhaps the most natural proof procedure for resolution is the level saturation procedure, where all resolvents at a certain level are determined before beginning the next level. Because we enumerate all resolvents eventually, we have a complete proof procedure. We define all the given, or input, clauses to be at level 0. A resolvent is at level $k+1$ if the parent clauses are at level k and j, for $j \leq k$. For the preceding example we list the clauses by level:

$$\begin{array}{ll}\text{Level 0:} & PaPb, Pa{\sim}Pb, {\sim}PaPb, {\sim}Pa{\sim}Pb \\ \text{Level 1:} & Pa, Pb, {\sim}Pa, {\sim}Pb, {\sim}PaPa, {\sim}PbPb \\ \text{Level 2:} & \Box \text{ , and all Level 0 and Level 1 clauses.}\end{array}$$

Level 2 resolvents include all previously obtained clauses because the tautologies $\sim PaPa$ and $\sim PbPb$ as one parent clause will always produce the other parent clause as the resolvent. Thus we can remove tautologies without affecting completeness. This is true for almost all resolution restrictions also. (An exception can occur when clauses are ordered, such as in the locking, or indexing, strategy. E.g., see Loveland [Lov78].)

Other deletion rules can be advantageous, given that only clause $\Box$ is sought. Once Pa is produced, resolving thereafter with $PaPb$ or $Pa{\sim}Pb$ is counterproductive because the resolvents again contain Pa. It is almost immediate that, when trying to derive the (length 0) clause $\Box$, there is no need to retain a clause that contains another existing clause as a subclause. Eliminating a clause $C_1 \vee C_2$ when C_2 also exists, for clauses C_1 and C_2, is known as subsumption elimination. This generalizes at the first-order level (noted later) and can get reasonably technical, with various options. This is studied in detail in Loveland [Lov78]. Also see Wos et al. [WOL91].

A variant of the level saturation proof procedure that has proved useful in practice employs a unit preference search strategy. (See Wos et al. [WCR64].) Here resolvents at higher levels than the current level being "saturated" are computed whenever one of the parents is a unit clause (one-literal clause). The intuition here is that the resolvent is always at least one literal less in cardinality ("shorter") than the longer parent clause, which is progress since the goal clause is the empty clause, having length 0. Except for merging, these unit clause resolutions must be the resolutions that occur near the end of refutations, so it makes sense to "look ahead" in this limited way.

Having considered the level-saturation resolution proof procedure, which is a breadth-first search ordering, it is natural to ask about depth-first search procedures. This is of particular interest to logic programmers who know that this is the search order for Prolog. We want to do this in a manner that preserves completeness. (Why it is desirable to preserve completeness is a non-trivial question, particularly at the first-order level where searches need not terminate and the combinatorial explosion of resolvent production is the dominant problem. Two points speak strongly for considering completeness: 1) these inference systems have such a fine-grained inference step that incompleteness in the inference proof procedures leads to some very simple problems being beyond reach and 2) it is best to understand what one has to do to maintain completeness to know what price must be paid if completeness is sacrificed. We will make some further specific comments on this issue later.)

Before presenting a depth-first proof procedure let us observe that at the propositional, or ground, level there are only a finite number of resolvents possible beginning with any given (finite) clause set. This follows directly from two facts: 1) no new atoms are created by the resolution inference rule, and 2) a literal appears at most once in a clause, so there is an upper bound on the number of literals in any resolvent.

A straightforward depth-first proof procedure proceeds as follows. If there are n input clauses then we temporarily designate clause n as the *focus* clause. In general the focus clause is the last resolvent still a candidate for inclusion in a resolution refutation (except for the first clause assignment, to clause n). We begin by resolving focus clause n against clause 1, then clause 2, etc. until a new non-subsumed non-tautological clause is created as a resolvent. This resolvent is labeled clause $n+1$ and becomes the new focus clause. It is resolved against clause 1, clause 2, etc. until a new clause is created that is retained. This becomes the new focus clause and the pattern continues. If $\Box$ is obtained the refutation is successful. Otherwise, some focus clause m creates no new surviving clause and *backtracking* is begun. The clause $m-1$ is relabeled the focus clause (but clause m is retained) and clause $m-1$ is resolved against those clauses not previously tried, beginning with clause $j+1$ where clause j is the clause that paired with clause $m-1$ to produce clause m. The first retained resolvent is labeled clause $m+1$. Clause $m+1$ now becomes the focus clause and the process continues as before. This backtracking is close to the backtracking employed by Prolog.

The above depth-first procedure does differ from that used in Prolog, however. The primary difference is that after one backtracks from a clause that yields no new resolvent, the clause is not removed. If we are interested in a complete procedure then we might need to consider using this "abandoned" clause C with clauses of higher index not yet defined, and hence not yet tried with C. Suppose hypothetically that a four clause given set has only the following resolution refutation (excluding renumbering): clause 3 and clause 4 create a resolvent (clause 5), clause 1 and clause 2 create a resolvent (clause 6), and $\Box$ is the resolvent of clause 5 and clause 6. The depth-first search starts its search by creating clause 5 as before, failing to create any new resolvent using clause 5, and backtracks by restarting with clause 4. Clause 4 first pairs with itself, but automatically skips this since no nontautologous clause resolves with itself, and fails when paired with clause 5. We have exhausted our first chosen focus clause; when this happens the backtracking continues backwards through all clauses. Thus, clause 3 is the new focus clause, does create a resolvent when paired with clause 4, but that is subsumed by (is identical to) clause 5. Clause 3 subsequently fails as the focus clause and clause 2 becomes the focus clause, and produces a resolvent when paired with clause 1. This new clause 6 immediately becomes the focus clause, is eventually paired with clause 5 and $\Box$ is produced. The point of all this is that clause 5 had to be retained to be used when clause 6 was finally created.

The reader may object to the supposition that clause 5 fails to produce any resolvent, but the contention that it would always pair with a lower indexed clause if it is ever used in a refutation would require a theorem about behavior of resolution refutations. The depth-first search utilized here is simply designed to sweep the same proof space as swept by the breadth-first search, but in a different order.

It does happen that clauses that exhaust their pairings and force a backtrack do not need to be retained. (I.e. clause 5 above would either yield a resolvent under some pairing or could be removed upon backtracking.) This is a consequent of the existence of linear resolution refinements. The linear refinement asserts that if a clause set is unsatisfiable then there is a resolution refutation such that each resolvent has the preceding clause as one of the parent clauses and the other parent clause must appear earlier in the deduction or be an input clause. The preceding parent clause is called the *near parent* clause. The other clause, the *far parent* clause, can be defined more precisely as follows: The far parent clause must be either an input clause or an *ancestor* clause, where the set of ancestor clauses of a clause C is the smallest set of clauses containing the near parent clause of C and the ancestors of the near parent clause of C. Again note that this is a condition on proof form, not on search. There exists a linear refutation with any input clause as first near-parent clause if that clause is in a minimally unsatisfiable subset of the input clauses. We call the first near-parent clause the *top* clause of the linear deduction. We give a linear refutation of the clause set given earlier. It is common to list the top clause as the last input clause.

A linear refutation:

1.	$PaPb$	input clause
2.	$Pa{\sim}Pb$	input clause
3.	${\sim}PaPb$	input clause
4.	${\sim}Pa{\sim}Pb$	input clause
5.	${\sim}Pb$	resolvent of 2, 4
6.	Pa	resolvent of 1,5
7.	Pb	resolvent of 3,6
8.	$\square$	resolvent of 5,7

The last step in the above refutation involved two resolvents. This is true of every refutation of this clause set, since no input clause is a one-literal clause and $\square$ must have two one-literal parents. This provides an example that not every clause set has a linear refutation where the far parent is always a input clause. Clauses 4, 5 and 6 are the ancestors of clause 7 in the above example. This definition of ancestor omits clauses 1, 2 and 3 as ancestors of clause 7 but our intent is to capture the derived clauses used in the derivation of clause 7.

The linear restriction was independently discovered by Loveland (see [Lov70], where the name *linear* was introduced and a stronger restriction *s-linear* resolution also was introduced) and by Luckham (where the name *ancestor filter* was used; see [Luc70].) A proof of completeness of this restriction also is given in Chang & Lee [CL73] and Loveland [Lov78].

With this result we can organize our depth-first search to discard any clause when we backtrack due to failure of that clause to lead to a proof. It follows that we need only try a given resolvent with clauses of lower index than the resolvent itself and that all retained resolvents are elements of the same (linear) proof attempt. Use of this restriction permits a great saving in space since only the current proof attempt clauses are retained. However, a large cost in search effectiveness may be paid because of much duplication of resolvent computation. In a proof search the same clause (or close variant) is often derived many times through different proof histories. In a breadth-first style search the redundant occurrences can be eliminated by subsumption check. This is a check not available when resolved clauses are eliminated upon backtracking as usually done in the depth-first linear resolution procedures. Each recreation of a clause occurrence usually means a search to try to eliminate the occurrence. This produces the high search price associated with depth-first search procedures.

The redundant search problem suffered by depth-first linear resolution procedures may make this approach unwise for proving deep mathematical theorems, where much computation is needed and few heuristic rules exist to guide the search. Depth-first linear resolution is often just the right procedure when there is a fairly strong guidance mechanism suggesting which clauses are most likely to be useful at certain points in the search. This principle is at work in the success of Prolog, which implements a depth-first linear strategy. When using Prolog, the user generally uses many predicate names, which means that relatively few literals are possible matches. This trims the branching rate. Secondly, the user orders the clauses with knowledge of how he expects the computation to proceed. Clearly, when some information exists to lead one down essentially correct paths, then there is a big win over developing and retaining all possible deductions to a certain depth. Besides logic programming applications, the depth-first approach is justified within artificial intelligence (AI) reasoning system applications where the search may be fairly restricted with strong heuristic guidance. When such conditions hold, depth-first search can have an additional advantage in that very sophisticated implementation architectures exist (based on the Warren Abstract Machine; see [AK90]) allowing much higher inference rates (essentially, the number of resolution operations attempted) than is realized by other procedure designs. This speed is realized for a restricted linear form called linear input procedures, which we discuss later, which is usually implemented using depth-first search mode.

It is important to realize that the linear resolution restriction need not be implemented in a depth-first manner. For example, one might begin with several input clauses "simultaneously" (either with truly parallel computation on a parallel computer or by interleaving sequential time segments on a sequential machine) and compute all depth n+1 resolvents using all depth n parents before proceeding to depth n+2 resolvents. This way subsumption can be used on the retained clauses (without endangering completeness) yet many resolution rule applications are avoided, such as between two input clauses not designated as start clauses for a linear refutation search.

The mode of search described above is very close to the set-of-support resolution restriction introduced by Wos et al. [WCR64, WRC65]. A refutation of a clause set S with set-of-support $T \subseteq S$ is a resolution refutation where every resolvent has at least one parent a resolvent or a member of T. Thus two members of $S - T$ cannot be resolved together. To insure completeness it suffices to determine that $S - T$ is a satisfiable set. The primary difference between this restriction and the quasi-breadth-first search for linear resolution described in the preceding paragraph is that for the linear refutation search of the previous paragraph one could choose to segregate clauses by the deduction responsible for its creation (labeled by initial parents), and not resolve clauses from different deductions. This reduces the branching factor in the search tree. If this is done, then any subsuming clause must acquire the deduction label of the clause it subsumes. The resulting refutation will not be linear if a subsuming clause is utilized, but usually the refutation existence, and not the style of refutation, is what matters. One could call such refutations *locally linear* refutations. Loveland [Lov78] discusses linear restrictions with subsumption.

Before concluding this discussion of search strategies, we consider oriented clause sets. So far, we have treated the clause set as non-oriented, in that no clause or subset of clauses received special notice *a priori*. Such is the situation when an arbitrary logical formula is tested for unsatisfiability.

However, often the clause set derives from a theorem of form $A \supset B$, where A is a set of axioms of some theory and B is the theorem statement believed to hold in the theory. Analogously, A may be viewed as a logic programming program and B is the query. It is well-known that testing the validity of $A \supset B$ answers the question: Is B a logical consequence of A?

We test $A \supset B$ for validity by testing $A \wedge \sim B$ for unsatisfiability. The conversion to Skolem conjunctive form of $A \wedge \sim B$ permits A and $\sim B$ to be converted separately and the two clause sets conjoined. The clause set S_A from A is called the theory base, database or axiom set, and the clause set S_B from $\sim B$ is called the goal set, query set or theorem statement (set). In practice, the cardinality of set S_A is usually much larger than the cardinality of S_B. Also, there is often no assurance that all the clauses of S_A are needed in the proof, whereas usually all clauses of S_B are needed. Moreover, S_A is usually a consistent clause set. Thus, the support set T is usually a subset of S_B, and linear refutations are started with one parent from S_B.

In this oriented clause set setting there are natural notions of direction of search. A linear refutation with one parent from S_B is called a *backchaining, goal-oriented*, or *top-down* refutation and the associated search is a *backchaining* or *goal-oriented* search. If the linear refutation begins with both parents from S_A then it is a *forward chaining, forward-directed*, or *bottom-up* refutation, with the same labels applied to the associated search. The same labels apply to the set-of-support refinement with support set $T \subset S_B$ or $T \subset S_A$. If top-down and bottom-up refutations are sought simultaneously, then the search is said to be *bidirectional*. The idea of bidirectionality is to share resolvents between the bottom-up and top-down searches, in effect having search trees growing bottom-up and top-down which meet and provide a complete proof path from data to goal. If this occurs, the search trees involved are roughly half the depth of a search tree strictly top-down or a tree strictly bottom-up. Since the number of tree nodes (resolvents) of any level generally grows exponentially with the depth of the level a tremendous potential savings exists in search size. Automated theorem provers use bidirectional search sometimes, but use is by no means universal. One problem is what clauses from S_A are appropriate. In back-chaining systems, often definitions are expanded forward several steps, a case of forward chaining.

A study of bidirectional search in a more general setting that resolution inference was done by Pohl [Poh71]. A related study on search strategies, explicitly considering resolution, is given in Kowalski [Kow70].

Before proceeding to further restrictions of resolution we need to present the first-order version of the resolution procedure. The more common presentation of first-order resolution uses two inference rules, binary resolution and binary factoring. Binary resolution is the first-order counterpart of the propositional resolution rule given earlier and factoring is the generalization of the merging of literals that removes duplication in the propositional case. ("Binary" refers to action limited to two literals; if factoring is incorporated within the resolution rule then several literals in one or both parent clauses might be "removed" from the resolvent.)

To present the binary resolution and factoring rules of inference, unification must be introduced. Substitution θ is a unifier of expressions l_1 and l_2 iff $l_1\theta \equiv l_2\theta$. For our use it suffices to consider l_1 and l_2 as (first-order) atoms of first-order terms. Substitution σ is a most general unifier (mgu) of l_1 and l_2 iff $l_1\sigma \equiv l_2\sigma$ and for any substitution γ such that $l_1\gamma = l_2\gamma$, there is a λ such that $\gamma = \sigma\lambda$. We defer to the chapter on unification in Volume 1 of this handbook for further discussion of this topic. Also see the chapter on resolution-based deduction in the same volume.

Two clauses are variable disjoint if they share no common variables.

We now present the inference rules for first-order resolution.

Binary resolution. Given variable disjoint clauses $C_1 \vee a_1$ and $C_2 \vee {\sim}a_2$, where a_1 and a_2 are atoms and C_1 and C_2 are disjunctions of literals, we deduce resolvent $(C_1 \vee C_2)\sigma$ from $C_1 \vee a_1$ and $C_2 \vee {\sim}a_2$, where σ is the mgu of a_1 and a_2; in summary:

$$\frac{C_1 \vee a_1 \quad C_2 \vee {\sim}a_2}{(C_1 \vee C_2)\sigma}$$

Binary Factoring. Given clause $C \vee l_1 \vee l_2$, where l_1 and l_2 are literals and C is a disjunction of literals, we deduce factor $(C \vee l)\sigma$ where σ is the mgu of l_1 and l_2, i.e. $l_1\sigma = l = l_2\sigma$; in summary:

$$\frac{C \vee l_1 \vee l_2}{(C \vee l)\sigma}$$

We illustrate the use of these rules by giving a refutation of the clause set

$$\{PxQx, Px{\sim}Qy, {\sim}PxQf(x), {\sim}Px{\sim}Qx\}$$

Because the resolution rule requires the two parent clauses to be variable disjoint, clause may have their variables renamed prior to the application of the resolution rule. This is done without explicit reference; we say that clause Px and clause ${\sim}PxRf(x)$ has resolvent $Rf(x)$ when the actual resolution rule application would first rename Px as (say) Py and then use

$$\frac{Py \quad {\sim}PxRf(x)}{Rf(x)}$$

We give a linear refutation:

1.	$PxQx$	input clause
2.	$Px{\sim}Qy$	input clause
3.	${\sim}PxQf(x)$	input clause
4.	${\sim}Px{\sim}Qx$	input clause
5.	${\sim}Qx{\sim}Qy$	resolvent of 2, 4
6.	${\sim}Qx$	factor of 5
7.	Px	resolvent of 1, 6
8.	$Qf(x)$	resolvent of 3,7
9.	$\Box$	resolvent of 6,8

The soundness of resolution procedures at the first-order level, that $\Box$ is derived only when the input clause set is unsatisfiable, is an immediate consequence of the fact that resolution and factoring preserve validity. The completeness of resolution procedures, that $\Box$ is derivable whenever the input clause set is unsatisfiable, utilizes the Skolem-Herbrand-Gödel theorem (a model-theoretic version of a very difficult proof-theoretic theorem of Herbrand). The Skolem-Herbrand-Gödel theorem asserts that a first-order formula (resp., clause set) is unsatisfiable iff a finite conjunction of ground formula (resp. clause set) instances is unsatisfiable. By a ground formula instance we mean a copy of the formula with the variables replaced by variable-free terms composed of symbols from the (non-variable) term alphabet of the formula. The set of such terms is called the *Herbrand Universe*. (If no constant individual appears in the formula, one is included by decree.) The standard completeness proof for a first-order resolution proof procedure has two parts: 1) a proof of the ground version of the procedure and 2) a "lifting" argument that relates the first-order search to a ground-level search by establishing that for any ground deduction of that class of deductions defined by the proof procedure there is a first-order deduction in that clause with that ground deduction as a ground instance. In particular, use of most general unifiers assures us that if clauses C_1 and C_2 have ground instances $C_1\theta$ and $C_2\theta$ in the ground deduction then the resolvent C_3 of C_1 and C_2 has $C_3\theta$ as a resolvent of $C_1\theta$ and $C_2\theta$. One can find more on the general nature of such arguments in Volume 1 of this handbook. Proofs of completeness of various restrictions of resolution must be argued separately; many such proofs appear in Chang & Lee [CL73] and Loveland [Lov78].

The search component of first-order proof procedures is as discussed for the propositional (or ground) procedures, with one notable exception. Since a set of resolvents generated from a finite set of clauses can be infinite, pure depth-first search may not terminate on some search paths. Rather, it always utilizes the last resolvent successfully and never backtracks. (This is well-known to Prolog users.) To insure that a refutation can be found when one exists, a variation of pure depth-first called iterative deepening is sometimes used. Iterative deepening calls for repeated depth-first search to bounds $d+nr$, for $d>0$, $r>0$ and $n=0,1,2,...$, where the user sets the parameters d and r prior to the computation. Frequently, $r=1$ is used. The advantage of small storage use, and speed for input linear search, is retained and yet some similarity to breadth-first search is introduced. The cost is recomputation of earlier levels. However, if one recalls that for complete binary trees (no one-child nodes and all leaves at the same level) there are as many leaves as interior nodes, and the proportion of leaves grows as the branching factor grows, one sees that recomputation is not as frightening as it first appears. Perhaps the more important downside to iterative deepening is that if you are structured so as to find the solution while sweeping a small portion of the entire search space with sufficient depth bound then a depth-bound just too low makes the procedure sweep the entire search space before failing and incrementing. This is a primary reason why Prolog employs pure depth-first search. (Also, of course, because very frequently Prolog does not have infinite search branches, at least under the clause ordering chosen by the user.)

Seeking linear refutations at the first-order level introduces a technical issue regarding factoring. The complication arises because we wish to enforce the notion of linearity strongly. In particular, if a factor of a far parent is needed that was not produced when that clause was a near parent, then the factor would have to be entered as a line in the deduction, violating the condition that the near parent always participates in creation of the next entry (by resolution or factoring). To avoid this violation we will agree that for linear resolution, unless otherwise noted, a resolution operation can include use of a factor of the far parent ancestor or input clause. Actually, for most versions of linear resolution, including the general form introduced already, this caveat is not needed provided that the given set has all its factors explicitly given also. (That is, the given set is closed under the factoring inference rule.) A fuller treatment of this issue, and much of what follows regarding linear resolution refinements and variants, is found in Loveland [Lov78].

1.2 Linear Resolution Refinements

There are several restrictions of linear resolution, and several combinations that multiply the possibilities for variations of linear resolutions. We will settle for one variant, among the most restricted of possibilities. Our interest in the particular form is that it is a resolution equivalent to the Model Elimination (SL-resolution) procedure that we study next.

We need to be more precise about the notion of clause subsumption before considering the next refinement. The idea of subsumption is to recognize when one formula makes a second formula redundant. This concept of subsumption can be expressed as follows: formula C subsumes formula D iff

$$\forall \overline{x} C \supset \forall \overline{y} D,$$

where $\overline{x}$ (resp. $\overline{y}$) denotes the free variables of C (resp. D). For example, $\forall x(P(x) \vee \sim P(f(x))$ implies $\forall y(P(y) \vee \sim P(f(f(y)))$; here, two instances of clause $Px{\sim}Pf(x)$ are needed to infer $\forall y(Py{\sim}Pf(f(y)))$, namely, $Px{\sim}Pf(x)$ and $Pf(x){\sim}Pf(f(x))$. Since we need an expedient test for subsumption, a form of subsumption used most often in resolution provers is a more limited form we call θ-subsumption: clause C θ-subsumes clause D if there exists a substitution θ such that the literals of $C\theta$ are also in D (we write this as $C\theta \subseteq D$, accepting the view of a clause as a set of literals), and that the number of literals of C does not exceed the number of literals of D. Without the literal count requirement every factor would be subsumed by its parent, and discarding clauses based on subsumption would generally not be possible. For many resolution strategies including breadth-first, linear and set-of support resolution, θ-subsumed clauses can be removed from further consideration.

We do not claim that the check for θ-subsumption always carries an acceptable cost; its use especially when C is a multiliteral clause has been controversial. However, strong use of θ-subsumption has been well defended (see [WOL91]). Certainly the test is much faster than the general definition because only one instance of C is involved.

We now introduce a small modification of the resolution inference rule that utilizes the θ-subsumption criterion. We say that C is an *s-resolvent* (for subsumption resolvent) of near parent clause C_1 and far parent clause C_2 if C is (a factor of) a resolvent C_3 where C θ-subsumes a substitution instance of near parent C_1. We amplify our reference to "substitution instance" below. Sometimes a factor of the standard resolvent is needed to satisfy the θ-subsumption restriction, and othertimes no s-resolvent exists. We will see that this highly constrains the resolvent; moreover, we can limit any resolution where the far parent is a non-input ancestor clause to s-resolution in the s-linear restriction we define shortly.

We give examples of s-resolution:

Example:

Near parent clause:	$PxQaRx$
Far parent clause:	$Qy{\sim}Rf(x)$
S-resolvent:	$Pf(x)Qa$
Standard resolvent:	$Pf(x)QaQy$

Example:

Near parent clause:	$PxQaRx$
Far parent clause:	$Qb{\sim}Rf(x)$
No s-resolvent exists	

The first example illustrates the reason we must allow s-resolvent C to θ-subsume only an instance of near parent clause C_1. In this example, the standard resolvent contains $Pf(x)$, created by the resolution operation from parent literal Px. Clearly, there is no instance of the resolvent that will θ-subsume near parent clause $PxQaRx$, but instance $Pf(x)QaRf(x)$ is θ-subsumed by $Pf(x)Qa$, here a factor of the standard resolvent.

The s-resolution operation can be quite costly to perform. The direct approach involves computing the resolvent and checking the θ-subsumption condition. The latter would most often cause rejection, so *a priori* tests on the parents would be desirable. We do not pursue this further because we finesse this issue later.

In pursuit of the strongly restricted form of linear resolution, we introduce the notion of ordered clause, with the intention of restricting the choice of resolving literal. By decreasing the choice we decrease the branching factor of the search tree. We set the convention that the rightmost literal of a clause is to be the resolving literal; however, we merge leftward, so that in the resolution operation the substitution instance demanded by the unifier of the resolution operation may create a merge left situation where the intended resolving literal disappears from the rightmost position! Then we say that such a resolution operation application fails.

The class of orderings for the restricted linear resolution we define permits any ordering the user chooses for input clauses except that each literal of each input clause must be rightmost in some ordered clause. Thus a 3-literal input clause must have at least three ordered clauses derived from it. A resolvent-ordered clause has leftmost the surviving descendents of the literals of the near parent ordered clause, in the order determined by the near parent ordered clause, while the surviving descendents of the far parent can be ordered in any manner the user wishes, i.e. is determined by the particular ordering the user has chosen.

The ordering applied to the input clauses is superfluous, except to the input clause chosen as first near parent clause (called the *top* clause), because all other uses of input clauses are as far parent clauses in a resolution operation. By definition of resolvent ordered clause, the literals descendent from the far parent can be ordered as desired when determining the resolvent. Of course, it is important that any ordering of input clauses provide that every literal of every input clause be rightmost, and thus accessible as a resolving literal. The notion of "top clause" is often used with arbitrary linear deductions, but is needed for ordered clause linear deductions because the first resolution step is not symmetrical in the parent clauses; one clause provides the leftmost literals of the ordered resolvent.

An *ordered s-linear deduction* of clause A from a set S of clauses is an ordered clause linear deduction of A from the fully factored set S_f of S where any resolvent without a input clause as a parent clause is an s-resolvent. If the fully factored input set is used then factors of ancestor clauses need not be derived in the deduction. This is true because it can be shown that factors of s-resolvents need not be computed.

Given a resolution refutation $B_1, \ldots, B_n$, it seems intuitive that if B_i θ-subsumes B_j, $i < j$, that we have made no progress towards $\Box$ in the interval $[i, j]$. We call a deduction *weakly tight* if for no $i < j$ is $B_i \subseteq B_j$. Full tightness, that for no $i < j$ does B_i θ-subsume B_j, is the ideal condition to enforce, but it is not clear that it is enforceable. A careful definition excludes clauses from θ-subsuming their factors but this hints at the care needed. Also, since s-resolution results in a shorter clause than the near parent (resolving literal of the near parent is removed at least) its use is a win when available. A deduction satisfies the *weak subsumption rule* if an s-resolvent is used whenever both parent clauses are derived clauses and the near parent clause is variable-free. In particular this rule says that for the propositional case no consideration to resolution with input clauses is needed when an s-resolvent exists. The rule is called "weak" because the unconstrained demand that s-resolution be used when available yields an incomplete procedure. The reason is intuitive; s-resolution always works when the right terms are at hand, but s-resolution where free variables occur can lead to forcing a wrong instantiation.

An ordered s-linear deduction is a TOSS deduction (weakly tight ordered s-linear resolution with the weak subsumption rule) if the deduction is weakly tight and satisfies the weak subsumption rule.

S-linear deductions exist that are neither weakly tight nor satisfy the weak subsumption rule.

We give two examples of ordered s-linear (actually, TOSS) deductions. The first example is one of the simplest examples where factoring is required in almost all resolution formats. Any order of input clauses is permitted.

1.	$PxPy$	input
2.	$\sim Px \sim Py$	input, top clause
3.	$\sim PxPy$	resolvent of 1,2
4.	$\sim Px$	S-resolvent of 2,3
5.	Px	resolvent of 1,4
6.	$\square$	S-resolvent of 4,5

This refutation uses an s-resolution with a input clause as one parent. Here the far parent is the top clause. The top clause can be treated like a derived clause, so that if derived clause 3 above had been ground, s-resolution could be forced. Actually, in this case no alternative need be considered anyway, because if the resolvent has every literal simply a variant of its parent literal no instantiation has been eliminated by this action. That is, the weak subsumption rule can be extended to force s-resolution if the s-resolvent does not instantiate the literals inherited from the near parent clause.

The second example has a more dramatic instance of s-resolution and illustrates how regular factoring of resolvents can then be dropped. However, it is seen from this example that there is a possible price to pay for banning factoring in that a longer proof is required. It is controversial whether factoring is better included or excluded in linear deductions. It is a trade-off of a reduced branching factor versus shorter proofs and hence shorter proof trees. (Recall that Prolog avoids factoring; factoring within linear resolution is optional in the Horn clause domain.)

The clause order is determined by $P < Q < R$ in the following example.

Example

1.	Qx	input
2.	$PxRx$	input
3.	$\sim PxRf(y)$	input
4.	$\sim Px\sim Rx$	input
5.	$Px\sim Qy\sim Rz$	input, top clause
6.	$Px\sim QyPz$	resolvent of 2,5
7.	$Px\sim QyRf(z)$	resolvent of 3,6
8.	$Px\sim Qy\sim Pf(z)$	resolvent of 4,7
9.	$Px\sim Qy$	s-resolvent of 6,8
10.	Px	resolvent of 1,9
11.	$Rf(x)$	resolvent of 3,10
12.	$\sim Pf(x)$	resolvent of 4,11
13.	$\square$	s-resolvent of 10,12

By factoring at step 6, steps 7 and 8 would have been omitted. In essence, step 7 through step 9 are repeated at step 11 through step 13, each sequence "removing" Px. These redundant proof segments are common when factoring is not used.

The TOSS restriction is not the strongest restriction possible; one can superimpose a restriction on the choice of far parent of s-resolutions. The added restriction is based on a restriction introduced in Andrews [And76], called *resolution with merging.* A resolvent may often contain a literal with parenthood in both parent clauses, such as literal Pa in resolvent $PaPx$ with parent clauses $PaQx$ and $PaPx\sim Qx$. This occurs because identical literals are merged. At the general level, unifiable literals may remain separate in the resolvent, but factoring then permits their "merging." Thus, $PxQy$ and $Px\sim Qy$ yields resolvent $PxPz$ with factor Px. When such a factor contains a literal with parenthood in both parent clauses, mimicking the ground case, we call this a *merge factor* and the pertinent literal(s) *merge literal(s).*

An MTOSS deduction (a TOSS deduction with merging) is a TOSS deduction where s-resolution occurs only when the rightmost literal of the far parent is a descendent of a merge literal.

Although the MTOSS restriction is as constraining a principle as is known regarding limiting resolution inferences when both parent clauses are derived, the merge condition has not received much attention in implementation. Actually, the TOSS and even the s-resolution restriction have not received much attention regarding implementations explicitly. As suggested earlier, a strongly-related format has received much attention.

The focus of the preceding consideration of linear restrictions is to reduce the need to resolve two derived clauses together. Why is this interesting? For several (related) reasons. One of the problems in proof search is the branching factor, how many alternative inferences could be pursued at a given point in a partial deduction. Since all input clauses are deemed always eligible (at least for linear resolution) the branching factor is reduced by tightening conditions for resolving two derived clauses. A second reason is that the input clauses can be viewed as given operators. In a linear deduction the far parent clause can be seen as modifying the near parent clause, transforming the near parent clause to a new but related clause. S-resolution is pleasing in this regard because it says that one needs deviate from applying the given "operators" only when the near parent is modified by removal of the literal receiving current attention, the resolving literal. (The ordering previously specified for linear deductions is useful in this viewpoint because it creates a focus on a given literal and its descendents before considering another literal of the clause.) An implementation consideration for limiting derived-clause resolution is that one can precondition the input clauses to optimize the resolution operation when a input clause is used in the resolution. Indeed, this is what Prolog does; compilers exist based on this principle.

To the extent that the above points have validity then the ideal linear resolution is the *linear input* format, where every resolvent has a input clause as far parent clause. (The word "linear" in "linear input" is redundant as use of a input clause as one parent forces a deduction to be linear, but the term is common and makes explicit that input resolution is a linear resolution restriction.)

Linear input resolution has been shown earlier not to be complete for first-order logic, but it is complete over a very important subset. A formula is a *Horn formula* if it is in prenex normal form with its quantifier-free part (matrix) in conjunctive normal form, where each clause has at most one positive literal. A *Horn (clause) set* is a clause set where each clause has at most one positive literal. That every unsatisfiable Horn set has a linear input refutation is immediate from the completeness of linear resolution and the observation that if one takes a clause of all-negative literals (a *negative clause*) as top clause then every derived clause is a negative clause and resolution between derived clauses is impossible. A negative clause must exist in each unsatisfiable set, for otherwise a model consisting of positive literals exists. The completeness proof for linear resolution shows that a refutation exists using as top clause any clause that is in a minimally unsatisfiable subset of the input clause set.

How much wider a class than Horn sets can linear input resolution handle? Essentially, no wider class. It can be shown (see Loveland [Lov78]) that if ground clause set S has a linear input refutation then there is a possible renaming of literals such that S_1, a renaming of S, is a Horn set. A ground set S_1 is a renaming of S if S and S_1 differ only in that for some atoms the positive and negated occurrences of that atom have been interchanged. This is equivalent to renaming each atom of a selected set of atoms by a new negated atom at every occurrences and replacing $\sim\sim A$ by A. Thus the ground set

$$\{\sim PaPbPc\sim Pd, \sim Pb\sim Pd, Pd\sim Pe, \sim Pc\sim Pe, Pe, Pa\}$$

is not Horn, because the first clause has two positive literals, but does have a linear input refutation, as the reader can easily check. However, the set is convertible to a Horn set by interchanging Pa, Pb, Pc with their negations.

This theorem that ground clause sets with linear input refutations are renamable Horn sets need not hold at the general level. This is because a literal within a clause C has several ground instances, some instances requiring renaming and others not, and without finding the refutation (or at least the instances needed) it is not possible to decide whether or not to rename the literal occurrence within C.

Horn sets are considered further when we focus on logic programming. We move on with our review of related theorem-proving procedures.

Apparently the linear input format is desirable, and sophisticated and effective architectures for Prolog implementations have indeed affirmed its desirability, but we seem condemned by the theorem just considered to clause sets whose ground image is a renamable Horn set. Actually, the manner of overcoming this problem was discovered before linear resolution was understood as such. The means of having linear input format complete for all of first-order logic is to alter the underlying resolution structure.

The *Model Elimination* (*ME*) procedure (see [Lov68, Lov69, Lov78]), introduced almost simultaneously with the resolution procedure, uses the notion of *chain*, analogous to an ordered clause but with two classes of literals, *A-literals* and *B-literals*. The set of B-literals in a chain can be regarded as a derived clause as resolution would obtain. SL-resolution [KK71], also a linear input procedure, uses the ME device of two classes of literals to achieve completeness for first-order logic. In this respect it is not strictly a resolution procedure in spite of its name (it differs from ME primarily in introducing a factoring operation) so does not conflict with the relationship between Horn clause sets and linear input resolution as its label might suggest.

As is well-known to logic programming experts, the procedure supporting Prolog is a linear input procedure applied to Horn clauses, but adapted pragmatically after experimentation with SL-resolution. ME owes its current success to a series of implementations by Mark Stickel [Sti84, Sti88]; the implementations are collected under the name Prolog Technology Theorem Prover. Recently, parallel implementations of ME have appeared [BCLM91, SL90, AL91]. (Some impressive sequential implementations accompany the parallel versions; e.g., see [AS92, LSBB91].) These implementations have used the Warren Abstract Machine (WAM) architecture [War, AK90] so successful in implementing Prolog.

We present a summary of the Model Elimination and SL-resolution procedures. We present the ME procedure and then note the distinctions introduced by SL-resolution.

There is great similarity between TOSS deductions and ME deductions; for a proper variant of the TOSS procedure there is an isomorphism with ME. (The variant allows no merging or factoring.) For that reason we will present an example, first with a ground TOSS refutation and then with a corresponding ME refutation. This will be helpful in following the description of ME. Following the informal, simplified introduction to ME and the examples we give a formal, precise definition of (one version of) ME.

The Model Elimination procedure has two inference operations, extension and reduction. Extension corresponds closely to the resolution operation and reduction is related to factoring. Whereas the resolution operation deletes both resolving literals, the corresponding *extension* operation retains the "resolving" literal of the "near parent" chain, the preceding chain, but promotes the descendent of the literal from a B-literal to an A-literal. A-literals capture and retain all the ancestor information that is really needed, it turns out. (We will drop the reference to descendent and identify the successor literal with the parent, even when instantiation occurs. This will cause no confusion as it is natural to think dynamically of literals passing to newly derived chains in a procedural way.) The counterpart of s-resolution is the *reduction* operation (not related to "reduction" in logic programming parlance) where the rightmost B-literal is removed because it is identical (unifiable in the general case) to an A-literal to its left. Factoring is not needed in ME, as mentioned; reduction can be seen to incorporate factoring and even merging, so merging of ground literals is not done either. (Factoring is explicitly done in SL-resolution although not needed for completeness.) Chains are ordered, like ordered clauses, and the rightmost B-literal is always the literal acted upon. Any A-literal that becomes rightmost is dropped. The ME procedure we describe is called *weak ME* in [Lov78], and was the first ME procedure introduced [Lov68].

We now give our two related examples. We take our examples from [Lov78], p. 170.

Example: We choose the clause ordering for resolvents that is the reverse order of their presentation in the prefix. Recall that the ordering only affects literals from the far parent; literals from the near parent have the order of the near parent.

A TOSS refutation:

1.	$\sim Pa$	input
2.	$\sim Pc$	input
3.	$\sim PbPd$	input
4.	$\sim PbPc\sim Pd$	input
5.	$PaPbPc$	input, top clause
6.	$PaPb$	resolvent of 2,5
7.	$PaPd$	resolvent of 3,6
8.	$PaPc\sim Pb$	resolvent of 4,7
9.	$PaPc$	s-resolvent of 6,8
10.	Pa	resolvent of 2,9
11.	$\square$	resolvent of 1,10

Now we give the ME counterpart.

Example: The chain-ordering for derived chains is as for the preceding example, namely to extend with literals in reverse order to their appearance in the prefix, the input clause set.

An ME refutation:

1.	$\sim Pa$	input clause
2.	$\sim Pc$	input clause
3.	$\sim PbPd$	input clause
4.	$\sim PbPc\sim Pd$	input clause
5.	$PaPbPc$	input, top clause
6a.	$PaPb[Pc]$	extend with 2
	(A-literal is then deleted as it is rightmost)	
6b.	$PaPb$	
7.	$Pa[Pb]Pd$	extend with 3
8.	$Pa[Pb][Pd]Pc\sim Pb$	extend with 4
9.	$Pa[Pb][Pd]Pc$	reduce
	(A-literal Pb allows the reduction)	
10.	Pa	extend with 2
11.	$\square$	extend with 1

We now present the specifications for the (weak) Model Elimination procedure.

A chain is a sequence of literals, where each literal is in one of two classes, A-literal or B-literal. The input clauses define elementary chains, consisting of all B-literals. Each input clause must define at least one chain for each literal of the input clause, with that literal rightmost. (This provides that each literal of each input clause is accessible as a resolving literal.)

Not every literal sequence is a useful chain. We will use only admissible chains. A chain is *admissible* iff:

(1) complementary B-literals are separated by an A-literal;

(2) no B-literal is to the right of an identical A-literal;

(3) no two A-literals have the same atom; and

(4) the rightmost literal is a B-literal

The reason for condition (1) is that complementary B-literals not separated by an A-literal originate in the same clause, and the offending nonadmissible chain would incorporate a tautologous clause. This can arise even if the associated input clause that contributed those literals is not tautologous because the derived chain undergoes instantiation. Reason (2) comes from the procedure design, from one basic view of ME as using the A-literals to attempt to define a model of the clause and where each attempt at a model is eliminated in turn in the process of obtaining a refutation. No redundant entries in a model specification is needed and the B-literal could only become an A-literal later. Reason (3), that no identical or complementary A-literals need occur combines the just mentioned relationship to models and recognition that the reduction operation should have been used when the right A-literal was a B-literal in an earlier chain. Reason (4) reflects automatic removal of rightmost A-literals.

A (weak) Model Elimination refutation is a sequence of chains, beginning with a top chain selected from the input elementary chains, ending with $\Box$, where each successor chain is derived by extension or reduction defined below.

Each inference rule takes a single chain as the parent chain. The extension operation also uses an *auxiliary* chain, for our purposes always a input elementary chain.

Given an admissible parent chain K and a variable-disjoint auxiliary elementary chain K_1, chain K_2 is formed by the *extension* rule if the rightmost literals of K and K_1 can be made complementary literals by unifier σ. The derived chain K_2 is then formed by deleting the rightmost literal of $K_1\sigma$, promoting the rightmost literal of K to A-literal, and appending the B-literals of $K_1\sigma$ to the right of $K\sigma$ in any order chosen by a user-given ordering rule. If no literals of $K_1\sigma$ exist, all rightmost A-literals of $K\sigma$ are dropped.

Given an admissible parent chain K, chain K_2 is formed by the *reduction* rule if there exists an A-literal which can be made identical to the rightmost literal by unifier σ. The derived chain K_1 is then formed by deleting the rightmost literal of $K\sigma$ and any A-literals that would then become rightmost.

The ordering rule that dictates the order of appended B-literals from the auxiliary chain in extension is arbitrary. In SL-resolution this freedom to process in user-desired order is described by a selection function that selects from the B-literals to the right of the rightmost A-literal the literal to next use for extension. In ME this is captured by assuming an oracle ordering rule that places the literals in order that the user chooses for his/her selection sequence. Although B-literals between A-literals carry their order forward under both inference rules, clearly this is a descriptive and processing convenience only. The order can be changed at any time by rationalizing that the ordering rule could have made this new selection initially. No inference depends on nonrightmost B-literals to place any constraints on reordering. The selection function mechanism of SL-resolution makes this ordering freedom clear.

The ME procedure also possesses a lemma mechanism that allows derived elementary chains to be added to the set of input chains. This must be done with great selectivity because the auxiliary set can explode in size very quickly. Also, the ability to use the derived chain in the manner of the compiled input clauses, seems to be lost. It seems that only unit clause lemmas are worth retaining, and only a selected subset of those. The lemma device has recently proven itself in the METEOR implementation (see [AS92]).

To prove substantial mathematical theorems using a linear input proof procedure will take some substantial added devices. In spite of the very high inference rate achievable due to the effective implementation techniques that takes advantage of the fixed set providing the far parent clauses (chains), the tremendous number of redundant calculations overwhelm the search. Some manner of recording computed search subtrees when successful (lemmas) and when failed, is needed. Such a methodology, called caching, is being studied and has had some success (see [AS92]). Success with devices like caching are needed if linear input provers are to be useful in proving substantial mathematical theorems. Not only do procedures that employ a breadth-first search style keep clauses for subsumption use, but forward-chaining systems tend to create more ground literals because of a frequent property of axioms. Axioms are often Horn clauses and the positive literal often contains only variables shared with another literal. By resolving the positive literal last it is often a ground literal when resolved.

Although possibly not the optimal procedural format for proving mathematical theorems, the linear input format is excellent for situations where strong search guidance exists, as previously noted. One of the situations where such guidance is often possible is in the logic programming domain. We now explore this domain.

2 The Logic Programming Paradigm

Logic programming as a phrase has come to have two meanings. Many people familiar only with Prolog regard logic programming in a very narrow sense; the use of Horn clause logic in a manner implemented by Prolog. In its broadest sense logic programming is a philosophy regarding computation, asserting that deduction and computation are two sides of the same concept. This viewpoint is built on the following general observation: an answer that can be computed using a problem-specific algorithm can also be determined by a proof procedure presented with the axioms of the appropriate theory and the problem as a theorem to be proven. In its fullest sense, then, logic programming is concerned with the development and use of logics whose proof procedures best fit our well-understood notion of good computation procedures. The advantage of good computation models for the proof search is that the user then understands the computational implications of the problem formulation as (s)he develops the problem specification.

In this section we will look at the notion of logic programming in its fullest sense, by a systematic presentation of the logical structures that preserve good computation properties. This is done by preserving the important procedural qualities possessed by Horn clause logic that makes Prolog so successful. Thus it is appropriate to begin by a review of logic programming in the narrow sense mentioned above, which will reflect the historical development of logic programming as a field of study. This will bring us back to the subject matter of the preceding section, for the inference engine that underlies Prolog is a restricted form of linear resolution using a revised notation.

2.1 Horn Clause Logic Programming

We first need to define *Horn clause logic.* If we recall that a clause is a disjunction of literals, then a *Horn clause* is a clause with at most one positive literal, or *atom.* Clauses with one atom are called *definite* clauses. One formulation of Horn clause logic is as a refutation logic of the type we have already studied with the clause set restricted to Horn clauses. Simple restrictions of refutation systems already considered give us complete refutation procedures for this logic. One restriction is to linear resolution; simply impose an input restriction on a standard linear resolution system. That is, the far parent clause must be from the input clause set. A simple observation extends the completeness of linear resolution for first-order logic to a completeness argument for linear input resolution over Horn logic. That is done as follows: Start with a negative clause (all literals are negative; there always is such a clause if the clause set is unsatisfiable) and then note that ANY linear deduction must also be an input deduction because every derived clause is a negative clause. (Since no two negative clauses can resolve with each other no two derived clauses can resolve together.) One can show that if a refutation exists then one exists starting with a negative clause. Thus, since in general an unsatisfiable clause set always has a linear refutation, there must be a refutation among the linear input deductions for each unsatisfiable Horn clause set. Finally, clauses can be ordered arbitrarily and resolution done on only the rightmost, or only the leftmost, literal of the current clause. This is clear because the subdeduction that "removes" that literal can only use input clauses, so is totally independent of the other literals of the clause.

A second approach to structure a linear procedure for Horn clause logic is to restrict the Model Elimination (SL-resolution) procedure by removing the reduction rule. This leaves only extension as an inference rule which makes no use of A-literals so the need for two types of literals has gone. If the A-literals are removed then extension is simply the binary resolution inference rule. Thus ME also collapses to a linear input resolution refutation procedure within the Horn clause domain. Again we note that if we begin with a chain from a negative clause then reduction is never called so that the above restriction is equivalent to the full ME procedure on the class of Horn clause sets. Thus this restriction is complete for Horn clause logic. This restriction clearly does not use factoring so we see that linear input resolution without factoring is complete for the Horn clause domain. In ME (SL-resolution) we can only order (select from) the most recently added literals. However, with only the extension rule of inference, we can reorder (select from) the entire clause at each step. Extension makes no reference to any literal within the current chain other than the literal extended on, so any literal may be chosen first. This restriction of ME (SL-resolution) to Horn clauses, with the changes noted, is called *SLD-resolution* for SL-resolution for Definite clauses. SLD-resolution was so-named by Apt and vanEmden for the reason outlined here (and, probably more importantly, because the development of Prolog's underlying logic, SLD-resolution, followed historically from SL-resolution). However, independent of the activity directly leading to Prolog, Robert Hill [Hil] defined and proved complete for Horn clause logic this ordered linear input procedure, which he called LUSH resolution, for Linear, Unrestricted Selection resolution for Horn clauses.

Horn clause logic provides the mechanism that defines "pure Prolog", the first-order logic part of Prolog. (We assume that the reader is familiar with the search control aspects of Prolog and with negation-as-failure, and knows that these are not practically definable within first-order Horn clause logic. Someone wishing further amplification of these qualities of Prolog should consult some of the other chapters of this volume.) To better capture the direct notion of proof and computation important to the presentation of Prolog we need to adopt another view of Horn clause logic. The new viewpoint says that we are not interested simply in whether or not a set of clauses is unsatisfiable. We now are interested in answers to a query made in the context of a database, to use one vocabulary common in logic programming. In a more compatible terminology, we seek a constructive proof of an existentially quantified theorem where the existential variables are instantiated so as to make the theorem true. (It happens that only one instantiation of these existential variables is needed for any Horn clause problem; this is a consequence of the need for only one negative clause in any (minimal) unsatisfiable Horn clause set, ground sets in particular.) Along with a notion of answer we will also emphasize a direct proof system, meaning that conceptually we will consider proofs from axioms with the theorem as the last line of the deduction. This is important even though our customary proof search will be a backward chaining system that begins with the proposed theorem (query).

The alternate viewpoint begins with a different representation for a Horn clause. The definite clauses have the generic form $A_1 \vee \sim A_2 \vee \sim A_3 \ldots \vee \sim A_n$ which we now choose to represent in the (classically) logically equivalent form $A_1 \leftarrow A_2, A_3 \ldots, A_n$ where the latter expression is the implication $A_2 \wedge A_3 \ldots \wedge A_n \supset A_1$ written with the consequent on the left to facilitate the clause presentation when backchaining in the proof search mode, and with the comma replacing AND. Negative clause $\sim A_1 \vee \sim A_2 \vee \sim A_3 \ldots \vee \sim A_n$ is written $\leftarrow A_1, A_2, A_3 \ldots, A_n$ which as an implication is read $A_1 \wedge A_2 \wedge A_3 \ldots \wedge A_n \supset FALSE$ (i.e., that the positive conjunction is contradictory). In particular, we note that the negative literals appear as atoms on the right of $\leftarrow$ (conjoined together) and the positive literal appears on the left.

We now consider the problem presentation. Theorem proving problems are often of the form $A \supset B$, where A is a set of axioms and B is the proposed theorem. The validation of $A \supset B$ is equivalent to the refutability of its negation, $A \wedge \sim B$. To assure that $A \wedge \sim B$ is a Horn clause set we can start with Horn clauses as axioms and let B have form $\exists \overline{x}(B_1 \wedge B_2 \wedge \ldots \wedge B_r)$, where $\exists \overline{x}$ means the existential closure of all variables in the expression it quantifies. This form works for B because $\sim \exists \overline{x}(B_1 \wedge B_2 \wedge \ldots \wedge B_r)$ is logically equivalent to a negative Horn clause. In Horn clause logic programming (i.e. for Prolog) the problem format is explicitly of the form $P \supset Q$, where P, the *program*, is a set of definite Horn clauses and Q, the *query*, is of form $\exists \overline{x}(B_1 \wedge B_2 \wedge \ldots \wedge B_r)$. The notation $\leftarrow B_1, \ldots, B_r$ that we use for this formula is suggestive of a query if the $\leftarrow$ is read here as a query mark. Indeed, ?- is the prefix of a query for most Prolog implementations. (The existential quantifier is not notated.)

We have just seen that the Prolog query is an existentially quantified conjunction of atoms. As already noted, it is central to logic programming that the proof search yield a substitution instance for each existentially quantified query variable and not just confirmation that the query instance follows from the program clause set. We call this *answer extraction.* Answer extraction is not original to logic programming; it goes back to work in question-answering systems of Green and Raphael [GR68, Gre69] for natural language systems and robotics in particular. Again, we note that for Horn clause sets only one query instance is needed; every Horn clause program implies (at most) a single query instance, a *definite* answer.

We emphasize that we have shifted from resolution theorem proving, which focuses on refutability and with a very symmetric input set (no distinguished literal or clauses) to a focus on logical consequence and input asymmetry. Asymmetry occurs at the clause level with a designated consequent atom and at the problem level with a designated query.

We now focus more closely on answer extraction. Given program P and query Q a *correct answer substitution* θ for P, Q is a substitution over Q such that $Q\theta$ is a logical consequence of P. For program P and query Q the *computed answer substitution* θ is the substitution applied to all variables of Q as determined at the successful termination of the SLD-resolution procedure. The key statements of soundness and completeness for Horn clause logic programming is that for SLD-resolution every computed answer is correct and if a correct answer exists then there exists a finite SLD-resolution deduction yielding a computed answer. Of course, this is a direct translation of results known to the resolution theory community. (We hasten to note that many important results in logic programming are not direct derivatives of work in the automated theorem proving field.)

Finally, we remark that the backchaining, problem reduction proof search mode inherited from linear resolution is an artifact of practicality only. The goal-oriented search has proved very advantageous, especially since user search control (e.g. ordering of clauses in the program) means that a small fraction of the search space is expanded in depth-first goal-oriented search. Conceptually, it is also useful to consider proofs in a forward direction, from facts through implied facts to the query instance.

We present a small example in the two formats of SLD-resolution to illustrate the isomorphism between the logic programming notation and the resolution notation for linear input deductions. We give the problem in the logic programming style first. Note that s, t, u, v, w, x, y, z are variables and a, b, c are constant terms. The substitutions that participate in the computed answer are indicated. The format is x/t where x is a variable and t is the replacement term.

Program:

1. $p(a, a, b)$
2. $p(c, c, b)$
3. $p(x, f(u, w), z) \leftarrow p(x, u, y), p(y, w, z)$
4. $p(x, g(s), y) \leftarrow p(y, s, x)$

Query:

5. $\leftarrow p(a, t, c)$

Derivation:

6. $\leftarrow p(a, u, y), p(y, w, c)$ using clause 3
   Substitution: $t/f(u, w)$
7. $\leftarrow p(b, w, c)$ using clause 1
   Substitution: u/a
8. $\leftarrow p(c, s, b)$ using clause 4
   Substitution: $w/g(s)$
9. success using clause 2
   Substitution: s/c

Answer:

$p(a, f(a, g(c)), c)$

That the deduction form is that of ordered clause input linear resolution is clear by comparing the above to the resolution deduction that follows.

Given clauses:

1. $p(a, a, b)$
2. $p(c, c, b)$
3. $p(x, f(u, w), z), \sim p(x, u, y), \sim p(y, w, z)$
4. $p(x, g(s), y) \sim p(y, s, x)$
5. $\sim p(a, t, c)$

Derivation:

6. $\sim p(a, u, y), \sim p(y, w, c)$ resolvent of 3,5
   Substitution: $t/f(u, w)$
7. $\sim p(b, w, c)$ resolvent of 1,6
   Substitution: u/a
8. $\sim p(c, s, b)$ resolvent of 4,7
   Substitution: $w/g(s)$
9. $\Box$ resolvent of 2,8
   Substitution: s/c

2.2 A Framework for Logic Programming

We have now completed our review of SLD-resolution and its role in Horn clause logic programming. Horn clause logic programming provides the core concepts underlying Prolog, the so-called pure Prolog. But is the quite restricted form given by Horn clause logic the only meaningful logic for logic programming? Do more powerful (and useful) logics exist? The idea of using arbitrary clauses as opposed to Horn clauses has been championed by [Kow79]. In a similar fashion, the use of full first-order logic — as opposed to logic in clausal form — has been advocated and explored [Bow82]. In a more conservative direction, extending the structure of Horn clauses by limited uses of connectives and quantifiers has been suggested. The best known extension of pure Horn clause logic within the logic programming paradigm permits negation in goals, using the notion of negation-as-failure. However, the idea of using implications and universal quantifiers and, in fact, arbitrary logical connectives in goals has also been advocated [GR84a, LT84, McC88a, McC88b, Mil89b, MNPS91].

There is a wide spectrum of logical languages between those given by Horn clause logic and full quantificational logic, especially if the derivability relation to be used is also open to choice. An obvious question that arises in this situation is whether some of these languages provide more natural bases for logic programming than do others. We argue for an affirmative answer to this question in this subsection. In particular, we describe a criterion for determining whether or not a given predicate logic provides an adequate basis for logic programming. The principle underlying this criterion is that a logic program in a suitable logical language must satisfy dual needs: it should function as a specification of a problem while serving at the same time to describe a programming task. It is primarily due to the specification requirement that an emphasis is placed on the use of symbolic logic for providing the syntax and the metatheory of logic programming. The programming task description requirement, although less well recognized, appears on reflection to be of equal importance. The viewpoint we adopt here is that the programming character of logic programming arises from thinking of a program as a description of a search whose structure is determined by interpreting the logical connectives and quantifiers as *fixed* search instructions.

From this perspective, the connectives and quantifiers appearing in programs must exhibit a duality between a search-related interpretation and a logical meaning. Such a duality in meaning cannot be attributed to these symbols in every logical language. The criterion that we describe below is, in essence, one for identifying those languages in which this can be done.

To provide some concreteness to the abstract discussion above, let us reconsider briefly the notions of programs and queries in the context of Horn clause logic. As has previously been noted, one view of a query in this setting is as a formula of the form $\exists \overline{x}\,(p_1(\overline{x}) \wedge \ldots \wedge p_n(\overline{x}))$; $\overline{x}$ is used here to denote a sequence of variables and $p_i(\overline{x})$ represents an atomic formula in which some of these variables appear free. The SLD-resolution procedure described in the previous subsection tries to find a proof for such a query by looking for specific substitutions for the variables in $\overline{x}$ that make each of the formulas $p_i(\overline{x})$ follow from the program. This procedure can, thus, be thought to be the result of assigning particular search interpretations to existential quantifiers and conjunctions, the former being interpreted as specifications of (infinite) OR branches with the branches being parameterized by substitutions and the latter being interpreted as specifications of AND branches. This view of the logical symbols is of obvious importance from the perspective of programming in Horn clause logic. Now, it is a nontrivial property of Horn clause logic that a query has a constructive proof of the kind described above if it has any proof at all from a given program. It is this property that permits the search-related interpretation of the logical symbols appearing in queries to co-exist with their logical or declarative semantics and that eventually underlies the programming utility of Horn clauses.

Our desire is to turn the observations made in the context of Horn clause logic into a broad criterion for recognizing logical languages that can be used in a similar fashion in programming. However, the restricted syntax of logic in clausal form is an impediment to this enterprise and we therefore enrich our logical vocabulary before proceeding further. In particular, we shall allow the logical symbols $\wedge$, $\vee$, $\supset$, $\forall$, $\exists$, $\top$ and $\bot$ to appear in the formulas that we consider. The first five of these symbols do not need an explanation. The symbols $\top$ and $\bot$ are intended to correspond to truth and falsity, respectively. We shall also use the following syntactic variables with the corresponding general connotations:

$\mathcal{D}$	A set of formulas, finite subsets of which serve as possible programs of some logic programming language.
$\mathcal{G}$	A set of formulas, each member of which serves as a possible query or goal for this programming language.
A	An atomic formula excluding $\top$ and $\bot$.
D	A member of $\mathcal{D}$, referred to as a program clause.

G	A member of $\mathcal{G}$, referred to as a goal or query.
$\mathcal{P}$	A finite set of formulas from $\mathcal{D}$, referred to as a (logic) program.

Using the current vocabulary, computation in logic programming can be viewed as the process of constructing a derivation for a query G from a program $\mathcal{P}$. The question that needs to be addressed, then, is that of the restrictions that must be placed on $\mathcal{D}$ and $\mathcal{G}$ and the notion of derivation to make this a useful viewpoint.

Towards answering this question, we shall describe a proof-theoretic criterion that captures the idea of computation-as-search. The first step in this direction is to define the search-related semantics that is to be attributed to the logical symbols. This may be done by outlining the structure of a simple nondeterministic interpreter for programs and goals. This interpreter either succeeds or does not succeed, depending on the program $\mathcal{P}$ and the goal G that it is given in its initial state. We shall write $\mathcal{P} \vdash_O G$ to indicate that the interpreter succeeds on G given $\mathcal{P}$; the subscript in $\vdash_O$ signifies that this relation is to be thought of as the "operational" semantics of an (idealized) interpreter. The behavior of this interpreter is characterized by the following search instructions corresponding to the various logical symbols; the notation $[x/t]G$ is used here to denote the result of substituting t for all free occurrences of x in G:

SUCCESS	$\mathcal{P} \vdash_O \top$.
AND	$\mathcal{P} \vdash_O G_1 \wedge G_2$ only if $\mathcal{P} \vdash_O G_1$ and $\mathcal{P} \vdash_O G_2$.
OR	$\mathcal{P} \vdash_O G_1 \vee G_2$ only if $\mathcal{P} \vdash_O G_1$ or $\mathcal{P} \vdash_O G_2$.
INSTANCE	$\mathcal{P} \vdash_O \exists x\, G$ only if there is some term t such that $\mathcal{P} \vdash_O [x/t]G$.
AUGMENT	$\mathcal{P} \vdash_O D \supset G$ only if $\mathcal{P} \cup \{D\} \vdash_O G$.
GENERIC	$\mathcal{P} \vdash_O \forall x\, G$ only if $\mathcal{P} \vdash_O [x/c]G$, where c is a constant that does not appear in $\mathcal{P}$ or in G.

These instructions may be understood as follows: The logical constant $\top$ signifies a successfully completed search. The connectives $\wedge$ and $\vee$ provide, respectively, for the specification of an AND and a nondeterministic OR node in the search space to be explored by the interpreter. The quantifier $\exists$ specifies an infinite nondeterministic OR node whose branches are parameterized by the set of all terms. Implication instructs the interpreter to augment its program prior to searching for a solution to the consequent of the implication. Finally, universal quantification is an instruction to introduce a new constant and to try to solve the goal that results from the given one by a (generic) instantiation with this constant.

Certain comments about the search instructions are in order before we proceed further. First of all, it is necessary to counter a possible impression that their structure is arbitrary. If $\vdash_O$ is interpreted as derivability in almost any reasonable logical system, then the conditions pertaining to it that are contained in the instructions AND, OR, INSTANCE, AUGMENT and GENERIC would be true in the reverse direction. Thus, in order to maintain a duality between a logical and a search-related reading of the logical symbols, the operational interpretation of these symbols is forced to satisfy the listed conditions in this direction. The instructions provided above represent, in this sense, a strengthening of the logical character of the relevant connectives and quantifiers in a way that permits a search-related meaning to be extracted for each of them. Further, the search interpretations that are so obtained are, quite evidently, of a natural kind. The interpretations for $\exists$ and $\wedge$ are, as we have observed, the ones accorded to them within the framework of Horn clauses, the notions of success and disjunctive search are meaningful ones, and some of the programming utility of the AUGMENT and GENERIC search operations will be discussed later in this section.

The second point to note is that we have addressed only the issue of the success/failure semantics for the various logical symbols through the presentation of the idealized interpreter. In particular, we have not described the notion of the *result* of a computation. There is, of course, a simple way in which this notion can be elaborated: existentially quantified goals are to be solved by finding instantiations for the quantifiers that yield solvable goals, and the instantiations of this kind that are found for the top-level existential quantifiers in the original goal can be provided as the outcome of the computation. However, in the interest of developing as broad a framework as possible, we have not built either this or any other notion of a result into our operational semantics. We note also that our interpreter has been specified in a manner completely independent of any notion of unification. Free variables that appear in goals are not placeholders in the sense that substitutions can be made for them and substitutions are made only in the course of instantiating quantifiers. Of course, a practical interpreter for a particular language whose operational semantics satisfies the conditions presented here might utilize a relevant form of unification as well as a notion of variables that can be instantiated. We indicate how such an interpreter might be designed in the next section by considering this issue in the context of specific languages described there.

In a sense related to that of the above discussion, we observe that our search instructions only partially specify the behavior to be exhibited by an interpreter in the course of solving a goal. In particular, they do not describe the action to be taken when an atomic goal needs to be solved. A natural choice from this perspective turns out to be the operation of backchaining that was used in the context of Horn clause logic. Thus, an instruction of the following sort may often be included for dealing with atomic goals:

ATOMIC $\quad$ $\mathcal{P} \vdash_O A$ only if A is an instance of a clause in $\mathcal{P}$ or $\mathcal{P} \vdash_O G$ for an instance $G \supset A$ of a clause in $\mathcal{P}$.

There are two reasons for not including an instruction of this kind in the prescribed operational semantics. First, we are interested at this juncture only in describing the manner in which the connectives and quantifiers in a goal should affect a search for a solution and the particular use that is to be made of program clauses is, from this perspective, of secondary importance. Second, such an instruction requires program clauses to conform to a fairly rigid structure and as such runs counter to the desire for generality in the view of logic programming that we are developing.

A notable omission from the logical connectives that we are considering is that of $\sim$. The interpretation of negation in most logical systems corresponds to an inconsistency between a proposition and a set of assumptions. In contrast, the natural search-related meaning for this symbol seems to be that of failure in finding a solution to a given goal. There is, thus, a considerable variance between the two desired views of the symbol. Further, the search-related view assumes that the abilities of the system of derivation can be circumscribed in some fashion. It appears difficult to represent this kind of an ability within a proof system and so we do not make an attempt to do this here. It is easy to provide for an alternative view of negation that is close to its logical interpretation as is done in [Mil89b]. There is, however, no need to add an explicit negation symbol for this purpose since it can be treated as defined in terms of $\supset$ and $\perp$. As for the symbol $\perp$ itself, there is a natural tendency to read it as failure. Once again, this does not correspond to the usual interpretation of $\perp$ within logical systems where it is considered to be a proposition that is "true" when there is a contradiction in the assumptions. In including an interpretation of $\perp$ that is close to its logical meaning, a choice has to be made between two competing views of the consequences of discovering a contradiction. The search instructions described above are compatible with either view. The view that is ultimately adopted is dependent on the logical system that is chosen to provide the declarative semantics; in particular, on which of the notions of provability that are described below is used.

The declarative semantics to be associated with the various connectives and quantifiers can be formalized by means of sequent-style proof systems. We digress briefly to summarize the central notions pertaining to such systems. The basic unit of assertion within these systems is that of a *sequent*. This is a pair of finite (possibly empty) sets of formulas $\langle\Gamma, \Theta\rangle$ that is written as $\Gamma \longrightarrow \Theta$. The first element of this pair is commonly referred to as the antecedent of the sequent and the second is called its succedent. A sequent corresponds intuitively to the assertion that at least one of the formulas in its succedent holds given as assumptions all those in its antecedent. (Although we shall not be concerned with the situation where Θ is empty, this usually corresponds to a contradiction in the assumptions.) A *proof* for a sequent is a finite tree constructed using a given set of *inference figures* and such that the root is labeled with the sequent in question and the leaves are labeled with designated *initial sequents*. Particular sequent systems are characterized by their choice of inference figures and initial sequents.

Figure 1 contains the inference figure schemata of interest in this paper. Actual inference figures are obtained from these by instantiating Γ, Θ and Δ by sets of formulas, B, C, and P by formulas, t by a term and c by a constant. There is, in addition, a proviso on the choice of constant for c: it should not appear in any formula contained in the lower sequent in the same figure. The notation B, Γ (Θ, B) that is used in the schemata is to be read as an abbreviation for $\{B\} \cup \Gamma$ $(\{B\} \cup \Theta)$ and a set of formulas can be viewed as being of this form even if $B \in \Gamma$ $(B \in \Theta)$. The initial sequents in the proof systems that we consider are of the form $\Gamma \longrightarrow \Theta$ where $\top \in \Theta$ or $\Gamma \cap \Theta$ contains either $\bot$ or an atomic formula.

Sequent-style proof systems of the kind we have described generally have three *structural* inference figure schemata, which we have not listed. Two of these schemata, usually called *interchange* and *contraction*, ensure that the order and multiplicity of formulas in sequents are unimportant in a situation where the antecedents and succedents are taken to be lists instead of sets. Our use of sets, and the interpretation of the notation B, Γ (Θ, B) that is explained above, obviates these schemata. The third kind of structural inference figure schema, commonly referred to as *thinning* or *weakening*, permits the addition of a formula to the antecedent or succedent of a sequent whose proof has already been constructed. Such inference figures are required when the antecedent and the succedent of an initial sequent are permitted to have at most one formula. Our choice of initial sequents, once again, obviates these inference figures.

We shall call an arbitrary proof that is obtained using the schemata in Figure 1 a **C**-proof. Placing additional restrictions on the use of the schemata in Figure 1 results in alternative notions of derivations. One such notion that is of interest to us in this paper is that of an **I**-proof. A proof of this kind is a **C**-proof in which each sequent occurrence has a singleton set as its succedent. As a further refinement, if we disallow the use of the inference figure schema $\bot$-R in an **I**-proof, we obtain the notion of an **M**-proof. We shall need these three notions of derivability later in this chapter and so we introduce special notation for them. We write $\Gamma \vdash_C B$, $\Gamma \vdash_I B$, and $\Gamma \vdash_M B$, if the sequent $\Gamma \longrightarrow B$ has, respectively, a **C**-proof, an **I**-proof, and an **M**-proof; if the set Γ is empty, it may be omitted from the left side of these three relations. The three relations that are thus defined correspond to provability in, respectively, classical, intuitionistic and minimal logic. More detailed discussions of these kinds of sequent proof systems and their relationship to other presentations of the corresponding logics can be found in [Fit69, Gen69, Pra65, Tro73]. It is relevant to note that in our presentation of these three provability relations we have excluded a structural inference figure schema called *cut*. As a result of this omission, the correspondence between our presentation and the customary definitions of the respective derivability relations depends on a fundamental theorem for the logics in question that is referred to as the cut-elimination theorem.

The framework of sequent systems provides a convenient means for formalizing the desired duality in interpretation for the logical connectives and quantifiers. The first step in this direction is

$$\frac{B, C, \Gamma \longrightarrow \Theta}{B \wedge C, \Gamma \longrightarrow \Theta} \wedge\text{-L} \qquad \frac{\Gamma \longrightarrow \Theta, B \qquad \Gamma \longrightarrow \Theta, C}{\Gamma \longrightarrow \Theta, B \wedge C} \wedge\text{-R}$$

$$\frac{B, \Gamma \longrightarrow \Theta \qquad C, \Gamma \longrightarrow \Theta}{B \vee C, \Gamma \longrightarrow \Theta} \vee\text{-L}$$

$$\frac{\Gamma \longrightarrow \Theta, B}{\Gamma \longrightarrow \Theta, B \vee C} \vee\text{-R} \qquad \frac{\Gamma \longrightarrow \Theta, C}{\Gamma \longrightarrow \Theta, B \vee C} \vee\text{-R}$$

$$\frac{\Gamma \longrightarrow \Theta, B \qquad C, \Gamma \longrightarrow \Delta}{B \supset C, \Gamma \longrightarrow \Theta \cup \Delta} \supset\text{-L} \qquad \frac{B, \Gamma \longrightarrow \Theta, C}{\Gamma \longrightarrow \Theta, B \supset C} \supset\text{-R}$$

$$\frac{[x/t]P, \Gamma \longrightarrow \Theta}{\forall x\, P, \Gamma \longrightarrow \Theta} \forall\text{-L} \qquad \frac{\Gamma \longrightarrow \Theta, [x/t]P}{\Gamma \longrightarrow \Theta, \exists x\, P} \exists\text{-R}$$

$$\frac{[x/c]P, \Gamma \longrightarrow \Theta}{\exists x\, P, \Gamma \longrightarrow \Theta} \exists\text{-L} \qquad \frac{\Gamma \longrightarrow \Theta, [x/c]P}{\Gamma \longrightarrow \Theta, \forall x\, P} \forall\text{-R}$$

$$\frac{\Gamma \longrightarrow \Theta, \bot}{\Gamma \longrightarrow \Theta, B} \bot\text{-R}$$

Figure 1: Inference Figure Schemata

to formalize the behavior of the idealized interpreter within such systems. We do this by identifying the notion of a *uniform proof*. This is an **I**-proof in which any sequent whose succedent contains a non-atomic formula occurs only as the lower sequent of an inference figure that introduces the top-level logical symbol of that formula. Suppose now that the sequent $\Gamma \longrightarrow G$ appears in a uniform proof. Then the following conditions must be satisfied with respect to this sequent:

- If G is $\top$, the sequent is initial.
- If G is $B \wedge C$ then the sequent is inferred by $\wedge$-R from $\Gamma \longrightarrow B$ and $\Gamma \longrightarrow C$.
- If G is $B \vee C$ then the sequent is inferred by $\vee$-R from either $\Gamma \longrightarrow B$ or $\Gamma \longrightarrow C$.
- If G is $\exists x\, P$ then the sequent is inferred by $\exists$-R from $\Gamma \longrightarrow [x/t]P$ for some term t.
- If G is $B \supset C$ then the sequent is inferred by $\supset$-R from $B, \Gamma \longrightarrow C$.
- If G is $\forall x\, P$ then the sequent is inferred by $\forall$-R from $\Gamma \longrightarrow [x/c]P$, where c is a constant that does not occur in the given sequent.

The structure of a uniform proof thus reflects the search instructions associated with the logical symbols. We can, in fact, define $\vdash_O$ by saying that $\mathcal{P} \vdash_O G$, *i.e.*, the interpreter succeeds on the goal G given the program $\mathcal{P}$, if and only if there is a uniform proof of the sequent $\mathcal{P} \longrightarrow G$. We observe now that the logical symbols exhibit the desired duality between a logical and a search-related meaning in exactly those situations where the existence of a proof within a given logical system ensures the existence of a uniform proof. We use this observation to define our criterion for establishing the suitability of a logical language as the basis for logic programming: letting $\vdash$ denote a chosen proof relation, we say that a triple $\langle \mathcal{D}, \mathcal{G}, \vdash \rangle$ is an *abstract logic programming language* just in case, for all finite subsets $\mathcal{P}$ of $\mathcal{D}$ and all formulas G of $\mathcal{G}$, $\mathcal{P} \vdash G$ if and only if $\mathcal{P} \longrightarrow G$ has a uniform proof.

2.3 Abstract Logic Programming Languages

Abstract logic programming languages as described in the previous subsection are parameterized by three components: a class of formulas each member of which can serve as a program clause, another class of formulas that corresponds to possible queries and a derivability relation between formulas. Within the framework envisaged, the purpose of the derivability relation is to provide a declarative semantics for the various logical symbols used. This purpose is realized if a well understood proof relation, such as the relation $\vdash_C$, $\vdash_I$, or $\vdash_M$, is used. Once a particular declarative semantics is chosen, it must be verified that this accords well with the desired operational or search semantics for the logical symbols. In general, this will be the case only when the permitted programs and queries are restricted in some fashion. In determining if the desired duality is realized in any given situation, it is necessary to confirm that the existence of a proof for a goal from a program entails also the existence of a uniform proof. If this is the case, then our logical language has a proof procedure that is goal-directed and whose search behavior is substantially determined by the logical symbols that appear in the goal being considered. Viewed differently, the defining criterion for an abstract logic programming language is one that ensures that a procedure of this kind may be used for finding proofs without an inherent loss of completeness.

Even though our criterion for identifying logical languages that can serve as the bases for logic programming seems to have content at an intuitive level, it is necessary to demonstrate that it actually serves a useful purpose. An important requirement in this regard is that it have a definite discriminatory effect. Towards showing that it does have such an effect, we first consider some recently proposed extensions to Horn clause logic and show that they fail to qualify as abstract logic programming languages. These extensions utilize the negation symbol that may be introduced into our setting by defining $\sim P$ to be $(P \supset \bot)$. With this understanding, the first non-example is one where $\mathcal{D}$ consists of the universal closures of atoms or formulas of the form $(B_1 \wedge \ldots \wedge B_n) \supset A$ where A is an atom and B_i is a literal, $\mathcal{G}$ consists of the existential closure of conjunctions of atoms and $\vdash$ corresponds to classical provability (see, for example, [Fit85]). Within the language defined by this triple, the set $\{p \supset q(a), \sim p \supset q(b)\}$ constitutes a program and the formula $\exists x\, q(x)$ corresponds to a query. This query is provable from the program in question within classical logic and the following is a **C**-proof for the relevant sequent:

$$\dfrac{\dfrac{\dfrac{p \longrightarrow \bot, p}{\longrightarrow \sim p, p}\supset\text{-R} \qquad \dfrac{q(a) \longrightarrow q(a)}{q(a) \longrightarrow \exists x\, q(x)}\exists\text{-R}}{p \supset q(a) \longrightarrow \sim p, \exists x\, q(x)}\supset\text{-L} \qquad \dfrac{q(b) \longrightarrow q(b)}{q(b) \longrightarrow \exists x\, q(x)}\exists\text{-R}}{p \supset q(a), \sim p \supset q(b) \longrightarrow \exists x\, q(x)}\supset\text{-L}.$$

There is, however, no uniform proof for $\exists x\, q(x)$ from the program being considered; such a proof would require $q(t)$ to be derivable from $\{p \supset q(a), \sim p \supset q(b)\}$ for some particular t, and this clearly does not hold. Thus, the language under consideration fails to satisfy the defining criterion for abstract logic programming languages. Another non-example along these lines is obtained by taking $\mathcal{D}$ to be the collection of (the universal closures of) positive and negative Horn clauses, $\mathcal{G}$ to consist of the existential closure of a conjunction of literals containing at most one negative literal and $\vdash$ to be classical provability [GR87]. This triple fails to be an abstract logic programming language since

$$\sim p(a) \vee \sim p(b) \vdash_C \exists x \sim p(x)$$

although no particular instance of the existentially quantified goal can be proved. For a final non-example, let $\mathcal{D}$ and $\mathcal{G}$ consist of arbitrary formulas and let $\vdash$ be provability in classical, intuitionistic or minimal logic. This triple, once again, does not constitute an abstract logic programming language, since, for instance,

$$p(a) \lor p(b) \vdash \exists x\, p(x)$$

regardless of whether $\vdash$ is interpreted as $\vdash_C$, $\vdash_I$ or $\vdash_M$ whereas there is no term t such that $p(t)$ is derivable even in classical logic from $p(a) \lor p(b)$.

To conclude that our criterion really makes distinctions, it is necessary to also exhibit positive examples of abstract logic programming languages. We provide examples of this kind now and the syntactic richness of these examples will simultaneously demonstrate a genuine utility for our criterion.

The first example that we consider is of a logical language that is slightly richer than Horn clause logic. This language is given by the triple $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$ where $\mathcal{G}_1$ is the collection of first-order formulas defined by the syntax rule

$$G ::= \top \mid A \mid G \land G \mid G \lor G \mid \exists x\, G$$

and $\mathcal{D}_1$ is similarly defined by the rule

$$D ::= A \mid G \supset A \mid D \land D \mid \forall x\, D.$$

As observed in Subsection 2.1, a positive Horn clause is equivalent in classical logic to a formula of the form $\forall \overline{x}\,(B \supset A)$ where B is a conjunction of atoms. This form is subsumed by the D formulas defined above. The queries within the Horn clause logic programming framework are of the form $\exists \overline{x}\, B$ where B is a conjunction of atoms. Formulas of this sort are, once again, contained in the set denoted by $\mathcal{G}_1$. It is thus evident that the paradigm of programming provided for by Horn clause logic is subsumed by the triple $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$ if indeed this turns out to be an adequate basis for logic programming. That this is the case is the content of the following proposition:

Proposition 1 *The triple $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$ constitutes an abstract logic programming language.*

The reader interested in a proof of this proposition is referred, amongst other places, to [MNPS91]. Although we do not present a proof here, it is instructive to consider the structure of such a proof. As noted already, the nontrivial part consists of showing that the existence of a classical proof guarantees also the existence of a uniform proof. There are two observations that lead to this conclusion in the case of interest. First, if $\mathcal{P}$ is a finite subset of $\mathcal{D}_1$ and G is a member of $\mathcal{G}_1$, then an inductive argument on the heights of derivations shows that the sequent $\mathcal{P} \longrightarrow G$ has a **C**-proof only if it has one in which (a) the inference figure $\bot$-R does not appear and (b) there is only one formula in the succedent of each sequent appearing in the proof. Second, in a derivation of the sort engendered by the first observation, the inference figures $\land$-L, $\supset$-L and $\forall$-L can be moved above $\wedge$-R, $\vee$-R and $\exists$-R if these immediately precede them. Thus, the derivation of the restricted form determined by the first observation can in fact be transformed into a uniform proof for the same sequent.

The outline of the proof for Proposition 1 brings out certain additional aspects that should be noted. To begin with, a derivation of the kind described in the first observation is obviously an **I**-proof and an **M**-proof. The content of this observation may thus be rephrased in the following fashion: the notions of provability in classical, intuitionistic and minimal logic are indistinguishable from the perspective of sequents of the form $\mathcal{P} \longrightarrow G$, assuming that $\mathcal{P}$ is a finite subset of $\mathcal{D}_1$ and G is an element of $\mathcal{G}_1$. The second point to note is that we are assured of the existence of a uniform proof regardless of whether the sequent in question has a proof in classical, intuitionistic or minimal logic. Thus, the triples $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$, $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_I \rangle$ and $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_M \rangle$ all determine the *same* abstract logic programming language.

We have observed that the programming paradigm based on Horn clauses is subsumed by the abstract logic programming language $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$. There is a correspondence in the converse direction as well. Using the equivalence of $G \supset A$ and $\sim G \vee A$, the operations of prenexing and anti-prenexing, de Morgan's laws and the distributivity of $\vee$ over $\wedge$, it can be seen that each formula in $\mathcal{D}_1$ is equivalent in classical logic to a conjunction of positive Horn clauses. Each formula in $\mathcal{G}_1$ can, in a similar fashion, be seen to be classically equivalent to the negation of a conjunction of negative Horn clauses. Now, it is easily observed that the union of a set S_1 of positive Horn clauses and a set S_2 of negative Horn clauses has a refutation only if S_1 augmented with some particular element of S_2 also has a refutation. Thus, the idea of solving a goal from a program in the abstract logic programming language $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$ can be reduced, by a process of translation, to logic programming using Horn clause logic. However, there are several advantages to not carrying out such a reduction and to utilizing the syntax for programs and goals embodied in the triple $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$ instead. First of all, preserving the richer syntax can lead to a more compact notation, given that the size of the conjunctive normal form of a formula can be exponentially larger than that of the formula itself. Second, this syntax allows for the explicit use of the symbols $\vee$ and $\exists$ in programs and goals with their associated search interpretations. These symbols provide useful programming primitives and also engender a syntax that is closer to the one used in practice; for instance, Prolog programmers often use disjunctions in the bodies of clauses with the intention of signifying a disjunctive search. Finally, the reduction described above depends significantly on the use of classical logic. The translation to clausal form is questionable if the abstract logic programming language under consideration is thought to be defined by the triple $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_I \rangle$ or the triple $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_M \rangle$ instead. This point is of particular relevance since the enrichment to this abstract logic programming language that we consider below is based on abandoning classical logic in favor of intuitionistic or minimal logic.

The language $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$ does not utilize the framework developed in the previous subsection fully. In particular, the symbols corresponding to the search operations AUGMENT and GENERIC are excluded from the syntax of goal formulas in this language. When we consider adding $\supset$ to the logical symbols already permitted in goals, we see that the use of classical logic to provide the declarative semantics does not accord well with the intended search interpretation for the various symbols. Consider, for example, the goal formula $p \vee (p \supset q)$. Given the desired search semantics for $\vee$ and $\supset$, we would expect an interpreter that attempts to solve this goal in the context of an empty program to fail; the interpreter should succeed only if either p is solvable from the empty program or q is solvable from a program containing just p, and clearly neither is the case. This expectation is manifest in the fact that the sequent $\longrightarrow p \vee (p \supset q)$ does not have a uniform proof. There is, however, a **C**-proof for this sequent as witnessed by the following derivation:

$$\cfrac{\cfrac{\cfrac{p \longrightarrow p, q}{\longrightarrow p, p \supset q}\ \supset\text{-R}}{\longrightarrow p \vee (p \supset q), p \supset q}\ \vee\text{-R}}{\longrightarrow p \vee (p \supset q)}\ \vee\text{-R.}$$

(The last inference figure in this derivation introduces a formula already in the succedent which need not be written, given our treatment of antecedents and succedents as sets.) The problem in this case arises from the fact that the formulas $p \vee (p \supset q)$ and $(p \supset p) \vee (p \supset q)$ are equivalent in classical logic. However, the search semantics of these two formulas are incompatible: the former permits the clause p to be used only in finding a solution for q, whereas the latter makes it available even in solving p. A problem of a similar nature arises when we consider interactions between $\exists$ and $\supset$. The "query" $\exists x\,(p(x) \supset q)$, for instance, has a classical proof from a program consisting of the clause $((p(a) \wedge p(b)) \supset q)$ although it has no derivation consistent with the search interpretation of $\exists$. In general, a derivability relation that is weaker than that of classical logic is needed for determining the declarative semantics of a logic programming language incorporating the AUGMENT search operation, and intuitionistic or minimal provability appear to be possible choices. It may appear somewhat paradoxical that a weaker derivability relation should provide the basis for a richer logic programming language. However, this apparent paradox disappears when we consider the equivalence of $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$, $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_I \rangle$ and $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_M \rangle$; intuitionistic and minimal derivability provide, in a certain sense, a tighter analysis of the declarative semantics of the same language.

We describe replacements for $\mathcal{D}_1$ and $\mathcal{G}_1$ that realize the richer syntax for programs and goals considered above. In particular, let G- and D-formulas be given now by the following mutually recursive syntax rules:

$$G ::= \top \mid A \mid G \wedge G \mid G \vee G \mid \forall x\, G \mid \exists x\, G \mid D \supset G,$$

$$D ::= A \mid G \supset A \mid \forall x\, D \mid D \wedge D.$$

We shall use $\mathcal{G}_2$ and $\mathcal{D}_2$ to refer to the classes of G- and D-formulas so defined. There is a correspondence between these D-formulas and those described by the logician Harrop [Har60, Tro73]. Assuming that the symbol B represents arbitrary formulas, the so-called *Harrop formulas* are equivalent in intuitionistic and minimal logic to the H-formulas defined by the rule

$$H ::= A \mid B \supset A \mid \forall x\, H \mid H \wedge H.$$

An interesting property of Harrop formulas, proved in [Har60], is the following: if $\mathcal{P}$ is a finite set of Harrop formulas and C is a non-atomic formula, then $\mathcal{P} \vdash_I C$ only if there is an **I**-proof of $\mathcal{P} \longrightarrow C$ in which the last inference rule introduces the logical connective of C. Thus, an **I**-proof of a sequent whose antecedent is a set of Harrop formulas can be made uniform "at the root." This observation does not in itself imply the existence of a uniform proof for $\mathcal{P} \longrightarrow C$ whenever it has an **I**-proof: there may be sequents in such a derivation whose antecedents contain formulas that are not Harrop formulas and whose proofs can therefore not be made uniform at the root. For example, consider the situation when $\mathcal{P}$ is empty and C is $(p \vee q) \supset (q \vee p)$. Following Harrop's observation, we see that a proof of the resulting sequent can be obtained from one for $p \vee q \longrightarrow q \vee p$. However, the antecedent of this sequent does not contain an H-formula and it is easily seen that a derivation for the sequent in intuitionistic logic cannot have the introduction of the disjunction in the succedent as its last step. Now, by further restricting the syntax of the H-formulas and of the formulas that we attempt to show follow from them, it is possible to ensure the applicability of Harrop's observation at each sequent in a derivation. This idea is, in fact, reflected in the definitions of the formulas in $\mathcal{G}_2$ and $\mathcal{D}_2$ above. Viewed differently, every subformula of a formula in $\mathcal{D}_2$ that appears in a positive context has the top-level structure of a Harrop formula. The members of $\mathcal{D}_2$ are, for this reason, referred to as *hereditary Harrop formulas.*

We shall show presently that the triple $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_I \rangle$ defines an abstract logic programming language. Before we do this, we illustrate the nature of programming in this language.

Example: The problem that we consider here is commonly referred to as the sterile jar problem. Assume that the following facts are given to us: a jar is sterile if every germ in it is dead, a germ in a heated jar is dead and a particular jar has been heated. The task is to represent this information so that the conclusion that there is some jar that is sterile can be drawn.

There is an obvious division of labor for the representation task at hand: the information that is provided is best represented by a program and the conclusion that is to be drawn can be thought of as a goal. Let *sterile* be a unary predicate corresponding to the sterility of jars, let $germ(X)$ and $dead(X)$ represent the assertions that X is a germ and that (the germ) X is dead, let $heated(Y)$ represent the fact that (the jar) Y has been heated and let $in(X, Y)$ correspond to the assertion that (the germ) X is in (the jar) Y. Letting j be a constant that denotes the given jar, the assumptions of the problem can be represented by the following formulas:

$\forall y\,((\forall x\,(germ(x) \supset (in(x,y) \supset dead(x)))) \supset sterile(y))$,
$\forall y\,\forall x\,((heated(y) \wedge (in(x,y) \wedge germ(x))) \supset dead(x))$, and
$heated(j)$.

It is easily seen that each of these formulas is a member of $\mathcal{D}_2$ and so they collectively constitute a program in the language under consideration. We shall denote this program by $\mathcal{P}$. An interesting aspect of these formulas is the translation into logical notation of the defining characteristic of sterility for jars. In determining that a given jar is sterile, it is necessary to verify that any germ that is in the jar is dead. Two forms of hypothetical reasoning arise in this context. First, we are interested exclusively in the germs that are in the jar and we only need to ascertain that these are dead. However, this property must hold of every germ hypothesized to be in the jar, not just those whose existence is known of explicitly at some point. These two requirements are reflected in the embedded universal quantifier and implications that appear in the first program clause above. Further, as the reader may observe from the discussion that follows, the programming interpretation for these two logical symbols and the use of the instruction ATOMIC described in the previous subsection yields exactly the desired "procedure" for determining the sterility of a jar.

The conclusion that we wish to draw can be represented by the formula

$\exists x\, sterile(x)$.

This formula is obviously a member of $\mathcal{G}_2$ and, hence, a goal in our extended language. We may think of reaching the desired conclusion by constructing a uniform proof for this goal from the program $\mathcal{P}$. Using the search interpretation of the existential quantifier, such a proof may be obtained by, first of all, constructing a proof for $sterile(j)$ from the same program. Note that we have exhibited clairvoyance in the choice of the instantiating term. Such foresight can, of course, not be expected of an actual procedure that looks for uniform proofs and we will discuss ways of dealing with this issue in an actual implementation. Now, the search instructions do not dictate the next step in finding a proof for $sterile(j)$. However, we may think of using the instruction ATOMIC for this purpose; as we shall see presently, ATOMIC turns out to be a derived rule in the context of constructing uniform proofs. Using this instruction with respect to the first program clause in the list above produces the (sub)goal

$\forall x\,(germ(x) \supset (in(x,j) \supset dead(x)))$

with the program still being $\mathcal{P}$. At this stage the instructions GENERIC and (two instances of) AUGMENT will be used, giving rise to the goal $dead(c)$ and the program $\mathcal{P} \cup \{germ(c), in(c,j)\}$, where c is assumed to be a new constant. The situation that has been produced reflects an attempt to show that jar j is sterile by showing that any (generic) germ that is assumed to be in it is dead. To proceed further, we may use ATOMIC with respect to the second clause in the list above, yielding the goal

$$(heated(j) \wedge (in(c, j) \wedge germ(c)));$$

once again, some foresight is needed in the choice of instance for the formula from the program. Invoking the instruction AND twice now results in the atomic goals $heated(j)$, $in(c, j)$, and $germ(c)$ that are to be solved from the program $\mathcal{P} \cup \{germ(c), in(c, j)\}$. Each of these goals can be immediately solved by using the ATOMIC instruction.

We could think of extracting a result from the uniform proof that has been constructed. This result can be the instantiation for the existentially quantified variable in the original goal that yields the uniform proof. Intuitively, this corresponds to exhibiting j as an example of a sterile jar.

The following proposition shows that the classes $\mathcal{D}_2$ and $\mathcal{G}_2$ constitute satisfactory replacements for $\mathcal{D}_1$ and $\mathcal{G}_1$ respectively. We present a proof of this proposition here to illustrate the kind of analysis involved in establishing such a property.

Proposition 2 *The triple $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_I \rangle$ constitutes an abstract logic programming language.*

Proof. Let $\mathcal{P}$ be a finite set of $\mathcal{D}_2$ formulas and G be a $\mathcal{G}_2$ formula. Since a uniform proof is also an **I**-proof, $\mathcal{P} \vdash_O G$ only if $\mathcal{P} \vdash_I G$. Thus we only need to show that the converse holds.

Let Γ be a finite subset of $\mathcal{D}_2$. We claim, then, that there cannot be an **I**-proof for $\Gamma \longrightarrow \bot$. This claim is established by an induction on the sizes of proofs, where the size of a proof is the number of sequents appearing in it. So suppose the claim is not true and let Γ be such that the derivation of $\Gamma \longrightarrow \bot$ is of least size. Now, the size of this derivation cannot be 1 since $\bot$ cannot be a member of Γ. If the size is greater than 1, then the possible structures for the formulas in $\mathcal{D}_2$ dictates that the last rule in the derivation must be one of $\supset$-L, $\forall$-L, or $\wedge$-L. In each of these cases, there must be a derivation of smaller size for a sequent $\Gamma' \longrightarrow \bot$ where Γ' is a set of $\mathcal{D}_2$ formulas. This contradicts our assumption about Γ.

Now let Δ be a finite subset of $\mathcal{D}_2$ and G' be a member of $\mathcal{G}_2$ and assume that $\Delta \longrightarrow G'$ has an **I**-proof of size l. We claim then that

(1) if $G' = G_1 \wedge G_2$ then $\Delta \longrightarrow G_1$ and $\Delta \longrightarrow G_2$ have **I**-proofs of size less than l,

(2) if $G' = G_1 \vee G_2$ then either $\Delta \longrightarrow G_1$ or $\Delta \longrightarrow G_2$ has an **I**-proof of size less than l,

(3) if $G' = \exists x\, G_1$ then there is a term t such that $\Delta \longrightarrow [t/x]G_1$ has an **I**-proof of size less than l,

(4) if $G' = D \supset G_1$ then $\Delta \cup \{D\} \longrightarrow G_1$ has an **I**-proof of size less than l, and

(5) if $G' = \forall x\, G_1$ then there is a constant c that does not appear in any of the formulas in $\Delta \cup \{\forall x\, G_1\}$ for which $\Delta \longrightarrow [c/x]G_1$ has an **I**-proof of size less than l.

This claim is proved, once again, by induction on the size of the derivation for $\Delta \longrightarrow G'$. If this size is 1, G' must be atomic and so the claim is vacuously true. For the case when the size is $s+1$, we consider the possibilities for the last inference figure. The argument is trivial when this is one of $\wedge$-R, $\vee$-R, $\exists$-R, $\supset$-R, and $\forall$-R. From the previous claim, this figure cannot be $\bot$-R. Given the structure of the formulas in Δ, the only remaining cases are $\forall$-L, $\wedge$-L and $\supset$-L. Consider the case for $\forall$-L, *i.e.*, when the last figure is of the form

$$\frac{[t/x]P, \Theta \longrightarrow G'}{\forall x\, P, \Theta \longrightarrow G'}$$

The argument in this case depends on the structure of G'. For instance, let $G' = \forall x\, G_1$. The upper sequent of the above figure is of a kind to which the inductive hypothesis is applicable. Hence, there is a constant c that does not appear in any of the formulas in $\Theta \cup \{[t/x]P, \forall x\, G_1\}$ for which $[t/x]P, \Theta \longrightarrow [c/x]G_1$ has an **I**-proof of size less than s. Adding below this derivation a $\forall$-L inference figure, we obtain an **I**-proof of size less than $s+1$ for $\forall x\, P, \Theta \longrightarrow [c/x]G_1$. Observing now that c cannot appear in $\forall x\, P$ if it is does not appear in $[t/x]P$, the claim is verified in this case. The analysis for the cases when G' has a different structure follows an analogous pattern. Further, similar arguments can be provided when the last inference figure is an $\wedge$-L or an $\supset$-L.

Now let $\mathcal{P} \longrightarrow G$ have an **I**-proof. It follows from the second claim that it must then have a uniform proof. The proof of the claim in fact outlines a mechanism for moving the inference figure that introduces the top-level logical connective in G to the end of the **I**-proof. A repeated use of this observation yields a method for transforming an **I**-proof of $\Gamma \longrightarrow G$ into a uniform proof for the same sequent.

□

The proof of Proposition 2 reveals a relationship between derivability in intuitionistic and minimal logic in the context of interest. In particular, let $\mathcal{P}$ be a finite subset of $\mathcal{D}_2$ and let G be a formula in $\mathcal{G}_2$. We have observed, then, that an **I**-proof of the sequent $\mathcal{P} \longrightarrow G$ cannot contain uses of the inference figure $\bot$-R in it. Thus any **I**-proof of such a sequent must also be an **M**-proof. In other words, these two notions of provability are indistinguishable from the perspective of existence of derivations for sequents of the form $\mathcal{P} \longrightarrow G$. It follows from this that $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_I \rangle$ and $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_M \rangle$ constitute the same abstract logic programming languages. We note that the introduction of implication together with its desired search interpretation leads to a distinction between classical provability on the one hand and intuitionistic and minimal provability on the other. It may be asked whether a similar sort of distinction needs to be made between intuitionistic and minimal provability. It turns out that a treatment of negation and an interpretation of the idea of a contradiction requires these two notions to be differentiated. We do not discuss this issue any further here, but the interested reader may refer to [Mil89b] for some thoughts in this direction.

We have raised the issue previously of what the behavior of the (idealized) interpreter should be when an atomic goal is encountered. We have also suggested that, in the case of the language $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_M \rangle$, the instruction ATOMIC might be used at such a point. Following this course is sound with respect to the defined operational semantics as the following proposition shows.

Proposition 3 *Let $\mathcal{P}$ be a finite subset of $\mathcal{D}_2$ and let A be an atomic formula. If A is an instance of a clause in $\mathcal{P}$ or if there is an instance $(G \supset A)$ of a clause in $\mathcal{P}$ such that $\mathcal{P} \longrightarrow G$ has a uniform proof, then $\mathcal{P} \longrightarrow A$ has an uniform proof.*

Proof. If A is an instance of a formula in $\mathcal{P}$, we can obtain a uniform proof of $\mathcal{P} \longrightarrow A$ by appending below an initial sequent some number of $\forall$-L inference figures. Suppose $(G \supset A)$ is an instance of a clause in $\mathcal{P}$ and that $\mathcal{P} \longrightarrow G$ has a uniform proof. We can then obtain one for $\mathcal{P} \longrightarrow A$ by using an $\supset$-L inference figure followed by some number of $\forall$-L below the given derivation and the initial sequent $A, \mathcal{P} \longrightarrow A$.

□

Using ATOMIC as a means for solving atomic goals in conjunction with the instructions for solving the other kinds of goals also yields a complete strategy as we now observe.

Proposition 4 *Let $\mathcal{P}$ be a finite subset of $\mathcal{D}_2$ and let A be an atomic formula. If the sequent $\mathcal{P} \longrightarrow A$ has a uniform proof containing l sequents, then either A is an instance of a clause in $\mathcal{P}$ or there is an instance $(G \supset A)$ of a clause in $\mathcal{P}$ such that $\mathcal{P} \longrightarrow G$ has an uniform proof containing fewer than l sequents.*

Proof. The following may be added as a sixth item to the second claim in the proof of Proposition 2 and established by the same induction: the sequent $\mathcal{P} \longrightarrow A$ has an **I**-proof containing l sequents only if either A is an instance of a clause in $\mathcal{P}$ or there is an instance $(G \supset A)$ of a clause in $\mathcal{P}$ such that $\mathcal{P} \longrightarrow G$ has an **I**-proof containing fewer than l sequents. The claim when embellished in this way easily yields the proposition.

□

The abstract logic programming language $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_I \rangle$ incorporates each of the search primitives discussed in the previous subsection into the syntax of its goals. It provides the basis for an actual language that contains two new search operations, AUGMENT and GENERIC, in addition to those already present in Prolog. At least one use for these operations in a practical setting is that of realizing scoping mechanisms with regard to program clauses and data objects. Prolog provides a means for augmenting a program through the nonlogical predicate called *assert* and for deleting clauses through a similar predicate called *retract*. One problem with these predicates is that their effects are rather global: an *assert* makes a new clause available in the search for a proof for every goal and a *retract* removes it from consideration in every derivation. The AUGMENT operation provides a more controlled means for augmenting programs: this operation makes program clauses available only during the search for solutions to *particular* goals. Consider, for instance, a goal whose structure is given schematically by

$$(D_1 \supset (G_1 \wedge (D_2 \supset G_2))) \wedge G_3,$$

where D_1 and D_2 represent formulas from $\mathcal{D}_2$ and G_1, G_2 and G_3 represent formulas from $\mathcal{G}_2$. Assume further that we are attempting to solve this goal from a program given by $\mathcal{P}$. The search interpretation for $\supset$ requires such a solution to be obtained by solving G_1 from the program $\mathcal{P} \cup \{D_1\}$, G_2 from the program $\mathcal{P} \cup \{D_1, D_2\}$ and G_3 from $\mathcal{P}$. The AUGMENT operation thus provides a means for making additions to a program in a well structured fashion. This idea can be exploited to realize block structuring in logic programming as well as to provide a (logically justified) notion of *modules* in this programming paradigm. The reader is referred to [Mil89b] and [Mil89a] for illustrative examples and for a general exploration of these abilities.

Just as the AUGMENT operation is useful in giving program clauses a scope, the GENERIC operation is useful in controlling the availability of objects. To understand this possibility, consider a goal that is given schematically by $\exists x\, ((\forall y\, G_1(x, y)) \wedge G_2(x))$. Solving this goal involves finding a term t such that for some "new" constant c the goals $G_1(t, c)$ and $G_2(t)$ are solvable. Note that the constant c is available only in the context of solving the goal $G_1(t, c)$. In particular, it cannot appear within t and thus cannot be transmitted into the context of solving $G_2(t)$ or "outwards" as an answer substitution. Further, while it can appear in terms created in the course of solving $G_1(t, c)$ and is, in this sense, available as an object in this context, it cannot be directly manipulated by the clauses that are utilized in solving this goal. The latter is a consequence of the fact that the quantifier $\forall y$ determines the *lexical* scope of c and thus controls the context in which it can be referred to directly. Viewed differently, the GENERIC operation provides a means for introducing a new constant whose lexical scope is determined by the symbol representing the operation and whose dynamic scope is the goal that is spawned as a result of the operation.

The GENERIC and AUGMENT operations can be used in tandem to realize the notion of abstract data types in logic programming. The essential idea that needs to be captured in this context is that of limiting direct access to particular objects to only those program clauses that are supposed to implement operations on them, while allowing these objects and the operations defined on them to be used in a larger context. To see how this requirement might be realized in our context, consider a goal of the form $\exists x\, \forall y\, (D(x, y) \supset G(x))$, where y does not appear in

$G(x)$. From one perspective, the variable y is a constant whose name is visible only in the program clauses contained in $D(x, y)$. Once this constant is "created," it can be used in the course of solving $G(x)$. However, the only way it can be referred to by name, and hence directly manipulated, in this context is by using one of the clauses in $D(x, y)$.

Although the above discussion provides the intuition guiding a realization of information hiding and of abstract data types in logic programming, a complete realization of these notions requires an ability to quantify over function symbols as well. This kind of a "higher-order" ability can be incorporated into the language we have presented without much difficulty. However, we do not do this here and refer the interested reader to [MNPS91] and [NM88] instead. We also note that a more detailed discussion of the scoping mechanism provided by the GENERIC operation appears in [Mil89a] and the language described there also incorporates a higher-order ability.

We consider now the issue of designing an actual interpreter or proof procedure for the abstract logic programming languages discussed in this subsection. Let us examine first the language $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$. The notion of a uniform proof determines, to a large extent, the structure of an interpreter for this language. However, some elaboration is required of the method for choosing terms in the context of the INSTANCE instruction and of the action to be taken when atomic goals are encountered. Propositions 3 and 4 and the discussion of SLD-resolution suggest a course that might be taken in each of these cases. Thus, if a goal of the form $\exists x\, G(x)$ is encountered at a certain point in a search, a possible strategy is to delay the choice of instantiation for the quantifier till such time that this choice can be made in an educated fashion. Such a delaying strategy can be realized by instantiating the existential quantifier with a "placeholder" whose value may be determined at a later stage through the use of unification. Placeholders of this kind are what are referred to as logic variables in logic programming parlance. Thus, using the convention of representing placeholders by capital letters, the goal $\exists x\, G(x)$ may be transformed into one of the form $G(X)$ where X is a new logic variable. The use of such variables is to be "cashed out" when employing ATOMIC in solving atomic goals. In particular, given a goal A that possibly contains logic variables, the strategy will be to look for a program clause of the form $\forall \bar{y}\, A'$ or $\forall \bar{y}\, (G \supset A')$ such that A unifies with the atom that results from A' by replacing all the universally quantified variables in it with logic variables. Finding such a clause results, in the first case, in an immediate success or, in the second case, in an attempt to solve the resulting instance of G. The interpreter that is obtained by incorporating these mechanisms into a search for a uniform proof is still nondeterministic: a choice has to be made of a disjunct in the context of the OR instruction and of the program clause to use with respect to the ATOMIC instruction. This nondeterminism is, however, of a more tolerable kind than that encountered in the context of the INSTANCE instruction. Moreover, a deterministic but incomplete procedure can be obtained by making these choices in a fixed manner as is done in the case of Prolog.

In devising an actual interpreter for $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_I \rangle$, we need also to think of a treatment of the search primitives embodied in $\supset$ and $\forall$. From inspecting the instructions corresponding to these symbols, it appears that relatively straightforward mechanisms can be used for this purpose. Thus, consider the situation when a goal of the form $D \supset G$ is encountered. We might proceed in this case by enhancing the program with the clause D and then attempting to solve G. Similarly, assume that we desire to solve the goal $\forall x\, G(x)$. We might think now of generating a new constant c and of attempting to solve the goal $G(c)$ that results from instantiating the universal quantifier with this constant.

The mechanisms outlined above can, in principle, be used to implement the new search primitives. However, some care must be taken to ensure that they mesh properly with the devices that are used in implementing the language $\langle \mathcal{D}_1, \mathcal{G}_1, \vdash_C \rangle$. One possible source of unwanted interaction is that between the use of logic variables to deal with INSTANCE and of new constants to realize GENERIC: since instantiations for existential quantifiers may be determined only after the introduction of new constants for dealing with universal quantifiers that appear within their scope, some method is needed for precluding the appearance of such constants in the instantiating terms. As a concrete example, consider solving the goal $\exists x\, \forall y\, p(x, y)$ from the program $\{\forall x\, p(x, x)\}$; we assume that p is a predicate symbol here. Following the strategy outlined up to this point in a naive fashion results in a success: the initial goal would be reduced to one of the form $p(X, c)$, and this can be unified with the (only) program clause by instantiating X to c. Notice, however, that this is an erroneous solution since the instantiation determined for X leads to a violation of the condition of newness on c at the time that it is generated.

To prevent incorrect solutions of the kind described above, it is necessary to limit possible instantiations for logic variables in some relevant manner. Some devices such as raising [Mil92], lifting [Pau87] and dynamic Skolemization [Fit90] have been proposed for this purpose. (Static Skolemization, the device generally used in the setting of classical logic, does not work in the current context.) These devices preserve the usual unification process but build the constraints that have to be satisfied into the "logic variable" that is used in connection with INSTANCE or the new "constant" that is introduced in connection with GENERIC. An alternative approach that is described in [Nad93] is to modify the naive treatment of INSTANCE and GENERIC only to the extent of including a numeric label with logic variables and constants and to constrain unification to respect the information contained in these labels. This approach requires much less bookkeeping than the earlier mentioned ones and is, in fact, simple enough to be incorporated into an abstract machine for the language described in this subsection [NJK93].

An adequate implementation of AUGMENT must consider two additional aspects. One of these arises from the fact that different programs might have to be considered at distinct points in a search. An efficient means is therefore needed for realizing changing program contexts. The additions to and the depletions from the program generally follow a stack discipline and can, in principle, be implemented using such a device. A complicating factor, however, is the imposition of determinism on the interpreter that we have outlined and the consequent need for backtracking when a failure is encountered along a solution path that is being explored. To understand why this might be a problem, let us consider a goal of the form $\exists x\,((D \supset G_1(x)) \wedge G_2(x))$ in the context of a program $\mathcal{P}$. Under the scheme being considered, the program would first be augmented with D and an attempt would be made to solve the goal $G_1(X)$, where X is a logic variable. Suppose that this attempt succeeds with a certain binding for X and also leaves behind an alternative (partially explored) solution path for $G_1(X)$. An attempt would now be made to solve the appropriate instance of $G_2(X)$ after removing D from the program. Assume, however, that the attempt to solve this instance of $G_2(X)$ fails. It is necessary, at this point, to continue the exploration of the alternative solution path for $G_1(X)$. However, this exploration must be carried out in the context of the appropriate program. In particular, all the clauses that were added in the course of trying to solve $G_2(X)$ must be removed and the clause D and others that might have been added along the alternative solution path for $G_1(X)$ must be reinstated. The kind of context switching that is required as a result of backtracking is, thus, fairly complex and some thought is required in devising a method for realizing it efficiently.

The second new aspect that needs to be dealt with in a satisfactory implementation of AUGMENT is that of program clauses containing logic variables. To see how such clauses might be produced, and also to understand what is involved in handling such clauses properly, let us consider the goal $\exists x\,(p(x) \supset g(x))$; we assume that p and g are predicate symbols here. Following the strategy already outlined, the attempt to solve this goal leads to an attempt to solve the goal $g(X)$ from a program that now also includes the clause $p(X)$. Notice that the symbol X that appears in these formulas is a logic variable and that this variable actually appears in the new program clause. The nature of this variable in the clause is different from that of the variables that usually appear in program clauses. While this variable can be instantiated, such an instantiation must be performed in a consistent way across all copies of the clause and must also be tied to the instantiation of X in the goal $g(X)$. Thus, suppose that the program from which we are attempting to solve the given goal contains the clauses

$\forall x\,((q(x) \wedge p(b)) \supset g(x))$, and
$(p(a) \supset q(b))$.

Using the first clause in trying to solve $g(X)$ produces the subgoals $q(X)$ and $p(b)$. The first subgoal can be successfully solved by using the second clause and the added clause $p(X)$. However, as a result of this solution to the first subgoal, the added clause must be changed to $p(a)$. This clause can, therefore, not be used to solve the second subgoal, and this ultimately leads to a failure in the attempt to solve the overall goal.

The problem with logic variables in program clauses occurs again in the system N-Prolog that is presented in the next section and is discussed in more detail there. An actual interpreter for the language $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_I \rangle$ that includes a treatment of several of the issues outlined here is described in [Nad93]. The question of implementing this interpreter efficiently has also been considered and we refer the interested reader to [NJK93] for a detailed discussion of this aspect. For the present purposes, it suffices to note that a procedure whose structure is similar to the one for constructing SLD-refutations can be used for finding uniform proofs for the extended language and that a satisfactory implementation of this procedure can be provided along the lines of usual Prolog implementations.

We have focussed here on using the criterion developed in the last subsection in describing abstract logic programming languages in the context of first-order logic. However, the framework that has been described is quite broad and can be utilized in the context of other logics as well. It can, for instance, be used in the context of a higher-order logic to yield a higher-order version of the language $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_I \rangle$. Such a language has, in fact, been described [MNPS91] and has provided the basis for a higher-order extension to Prolog that is called λProlog [NM88]. A discussion of some aspects of this extension also appears in the chapter on higher-order logic programming in this volume of the handbook. More generally, the framework can be utilized with a different set of logical symbols or with a different search interpretation given to the logical symbols considered here. Interpreted in this sense, our criterion has been used in conjunction with linear logic [HP91, HM94] and a calculus of dependent types [Pfe89] to describe logic programming languages that have interesting applications.

3 Extending the Logic Programming Paradigm

In the previous section we presented an abstract framework which attempts to capture the essence of the logic programming paradigm. We have seen that the Horn clause logic underlying Prolog fits this framework but that a much richer class of program clauses can be used if intuitionistic logic is employed in providing the underlying declarative semantics. Abstracting and formalizing the logic programming paradigm thus provides a mechanism to propose new logic programming languages. However, it also allows us to better understand logic programming languages developed independently of the framework. We illustrate this in this section with the consideration of two such existing logic programming languages.

One category of languages that we present, *N-Prolog* and the related *QN-Prolog*, extends Horn clause logic by permitting implication in goals and in the antecedent of program clauses. We shall see that this language is based on an abstract logic programming language. The proof procedures for N-Prolog and QN-Prolog may also be understood as ones that attempt to construct uniform proofs, and we illustrate this aspect in our presentation.

The second language presents a different situation, a datapoint seemingly outside the framework we have presented. The programming language family, called *near-Horn Prolog* (*nH-Prolog*), permits disjunction in the consequent of program clauses. This leads to some strikingly different characteristics relative to other logic programming languages, such as the possibility of disjunctive answers. (With a suitable encoding of classical negation one may view the logic as coextensive with full first-order logic.) The proof mechanism of one variant of nH-Prolog fits the structure of uniform proofs, a startling fact given the power of representation of the underlying logic. We show by illustration the key reasons why the strong proof restrictions hold for this variant.

The study of the nH-Prolog language we undertake here accomplishes two points simultaneously. First, it supports the claim that the nH-Prolog language is a very attractive way to approach the extended logic domain with regard to the logic programming paradigm. Second, understanding the relationship between the variant of nH-Prolog considered here and the notion of uniform proof enhances our understanding of both entities.

We begin our consideration of existing languages with N-Prolog.

3.1 A Language for Hypothetical Reasoning

The N-Prolog language, developed by Gabbay and Reyle [GR84b, Gab85] introduces the hypothetical implication. This mechanism allows exploration of the consequences of a hypothetical situation, where additions are temporarily made to the database, or program, when certain queries are addressed. The basic system is sound and complete for positive intuitionistic logic, and Gabbay has shown that it can be extended to a complete procedure for positive classical logic by adding one inference rule (a "restart" rule).

We first present the propositional form of N-Prolog. We follow closely Gabbay and Reyle [GR84b]. If P is a program and Q is a query, then $P\ ?\ Q$ expresses the problem of determining if query Q follows from program P. $P\ ?\ Q = 1$ denotes success in processing Q; $P\ ?\ Q = 0$ denotes finite failure. If we let $P + A$ denote $P \cup \{A\}$, where P is a program and A is a clause, then the key novel deductive step of N-Prolog is to define $P\ ?\ (A \supset B)$ as $(P + A)\ ?\ B$. That is, the conditional $A \supset B$ as query is interpreted as asking B in the context of P augmented by the hypothetical fact A. We now formalize propositional N-Prolog.

An *N-clause* is defined inductively as an atom or as

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n \supset B$$

where the A_i are N-clauses and B is an atom. An *N-program* or *database* is a set of N-clauses. An *N-goal* is any conjunction of N-clauses. For example, any Horn clause is an N-clause, as are $(((a \supset b) \supset c) \supset d)$ and $((a \supset b) \wedge c) \supset d$. The expression

$$(a \supset (b \supset c)) \supset ((a \supset b) \supset (a \supset c)),$$

an axiom in one formulation of intuitionistic logic, is not an N-clause, but the expression

$$((a \wedge b \supset c) \wedge (a \supset b) \wedge a) \supset c$$

that is intuitionistically equivalent to it is an N-clause. In general, formulas of form $A \supset (B \supset C)$ need to be rewritten in the intuitionistically equivalent form $A \wedge B \supset C$ in order to qualify as N-clauses.

There are three rules for computation for propositional N-Prolog.

(1) The rule for atoms has two parts: if Q is an atom then P ? Q succeeds iff

  (a) $Q \in P$ or

  (b) for some clause $A_1 \wedge \ldots \wedge A_k \supset Q$ in P we have that P ? $(A_1 \wedge \ldots \wedge A_k)$ succeeds.

(2) The rule for conjunctions states that P ? $(A_1 \wedge \ldots \wedge A_k)$ succeeds iff each P ? A_i succeeds.

(3) The rule for implication states that P ? $(A \supset B)$ succeeds iff $(P + A)$? B succeeds, where $P + A$ is the result of adding each conjunct of A to P.

To illustrate a deduction in N-Prolog we show that

$$P \text{ ? } ((a \wedge b \supset c) \wedge (a \supset b) \wedge a) \supset c$$

succeeds for $P = \emptyset$, the empty program.

Example: Show $\emptyset$? $((a \wedge b \supset c) \wedge (a \supset b) \wedge a) \supset c = 1$.

1. $\emptyset$? $((a \wedge b \supset c) \wedge (a \supset b) \wedge a) \supset c$
2. $a \wedge b \supset c, a \supset b, a$? c
3. $a \wedge b \supset c, a \supset b, a$? $a \wedge b$
4. Case 1: $a \wedge b \supset c, a \supset b, a$? a
   This case succeeds immediately.
5. Case 2: $a \wedge b \supset c, a \supset b, a$? b
6. $a \wedge b \supset c, a \supset b, a$? a
   This case now succeeds.

N-Prolog with quantifiers, which we might also call *general level N-Prolog* or *first-order N-Prolog*, addresses the extension of N-Prolog to the first-order level. The major complication in moving to the general level that is not shared by Prolog is the sharing of variables between program and goal. The task $P\ ?\ (a(X) \supset b(X))$ succeeds if $(P + a(X))\ ?\ b(X)$ succeeds, where variable X now is shared by program and goal. (We hereafter adopt the Prolog convention of capital letters for variables.)

To handle the sharing of variables, N-Prolog with quantifiers, hereafter *QN-Prolog*, uses two classes of variables, universal and existential. Here universal variables are denoted by U_i, for i an integer; existential variables use other letters of the alphabet, also possibly with subscripts. (Of course, a commercial system might permit arbitrary words in place of the subscript.) Atoms now are predicate letters with terms that may contain variables of the two different kinds. *QN-clauses*, *QN-goals* and *QN-programs* generalize from the propositional form in the obvious way except no universal variables are permitted in QN-goals. Recall that in Prolog variables in goals behave as existential variables, and as universal variables in the program, but now that universal and existential variables co-exist in programs, an explicit mechanism is necessary. Some explicit replacement of universal variables by existential variables is necessary, as illustrated below.

For a formal definition of QN-Prolog the reader should consult [GR84b]. We will use an example to illustrate the use of the two classes of variables. First, we emphasize the key attributes regarding variables use:

(1) Existential variables may be shared among program clauses and goals, reflecting the view that the existential quantifiers associated with the variables are exterior to the entire expression (i.e. $\exists\vec{X}(P\ ?\ Q)$);

(2) Only existential variables occur in goals.

We adapt Example E2 (Section 4) from [GR84b] for our illustration. Although somewhat contrived so as to present an interesting non-Horn structure in a compact example, the example does suggest that real databases (programs) exist where hypothetical implication occurs and thus processing can benefit from the use of QN-Prolog.

Example: (Example E2 in [GR84b].)
Assume that we are given the following database:

(d1) A person is neurotic if he/she is greatly disturbed when criticized by one of his/her friends.

(d2) Mary gets greatly disturbed when Sally criticizes her.

(d3) Sally criticizes everyone.

(d4) A person is a bad friend if there is someone who would become neurotic if this someone were befriended by the person.

The query is "Is there someone who is a bad friend?"

The database and query are formalized as follows:

(d1) $[f(U_1, U_2) \wedge cr(U_1, U_2) \supset d(U_2)] \supset n(U_2)$

(d2) $cr(s, m) \supset d(m)$

(d3) $cr(s, U_1)$

(d4) $[f(U_1, U_2) \supset n(U_2)] \supset b(U_1)$

(query) $b(Y)$

The query uses an existential variable, and all database clauses use universal variables.

We now follow [GR84b] in presenting one possible computation sequence for this database and query.

1.	d1,d2,d3,d4 ? $b(Y)$	
2.	d1,d2,d3,d4 ? $f(Y, X) \supset n(X)$	using d4
3.	d1,d2,d3,d4,$f(Y, X)$? $n(X)$	rule for implication
4.	d1,d2,d3,d4,$f(Y, X)$? $f(Z, X) \wedge cr(Z, X) \supset d(X)$	using d1
5.	d1,d2,d3,d4,$f(Y, X), f(Z, X), cr(Z, X)$? $d(X)$	rule for implication
6.	d1,d2,d3,d4,$f(Y, m), f(Z, m), cr(Z, m)$? $cr(s, m)$	using (d2)
7.	d1,d2,d3,d4,$f(Y, m), f(Z, m), cr(Z, m)$? $cr(s, m)$	using (d3).

In connection with this computation, it is interesting to note that in step 2 and in step 4 the universal variables U_2 and U_1 that appear in clauses d4 and d1 respectively are not instantiated during unification with the previous goal and have therefore to be renamed to an existential variable in the new goal. Note also that additions are made to the program in the course of computation that contain existential variables in them. An instantiation is determined for one of these variables in step 6 and this actually has the effect of changing the program itself.

Observe that the answer variable is uninstantiated; the bad friend exists, by the computation success, but is unnamed. The neurotic friend is seen to be Mary by inspection of the computation.

The derivation given by Gabbay and Reyle uses clause d3 at Step 7. Perhaps a more interesting derivation makes use of the augmented database at Step 7 and shows that clause d3 is unneeded. The intuition behind this alternative is that d1 says that for some instance of U_2 one can establish $n(U_2)$ by satisfying (an appropriate instance of) an implication, and d2 provides an appropriate implication for such an instance.

We now state the soundness and completeness result for QN-Prolog given in Gabbay [Gab85].

Proposition 5 *P ? Q succeeds in* QN-Prolog *if and only if*

$$\vdash \exists\vec{X}[(\forall\vec{U})P \supset Q]$$

holds in intuitionistic logic for program (database) P and query Q where $\vec{X}$ denotes all the existential variables in P and Q and $\vec{U}$ denotes all the universal variables in P.

The above proposition brings out the connection between QN-Prolog and the discussion in the previous section. QN-Prolog is readily seen to be based on an abstract logic programming language contained in the language $\langle \mathcal{D}_2, \mathcal{G}_2, \vdash_I \rangle$ that is described in Subsection 2.3. This language is in fact created from the Horn clause language by permitting formulas of the form $D \supset A$ to appear in goals and in the antecedents of program clauses where D is itself a conjunction of program clauses under this definition. To ensure the existence of uniform proofs, i.e. proofs in which the succedent is singleton and also atomic whenever an antecedent inference figure is used, the derivability relation that needs to be adopted is $\vdash_I$. To reinforce the understanding of QN-Prolog as an abstract logic programming language we present the uniform proof implicit in the previous example using the sequent calculus described in the previous section.

In the presentation of the proof we adopt several suggestions made in Subsection 2.3 to improve the abstract interpreter efficiency. We delay the instantiations by use of logic variables as suggested for the abstract interpreter and as executed in the description of QN-Prolog. When the proof is read from the bottom (conclusion) upward, existential variables replace the instances that the $\forall$-L inference figures would normally introduce. The use of these variables signifies that one, perhaps undetermined, instance of the clause has been created. Other instances may similarly be created from the explicitly quantified version of the clause that is present in the antecedent of the sequent.

Several nonstandard inference figures are used to aid the presentation. The inference figure CI (clause incorporation) contains a formula in the antecedent of the upper sequent whose universal closure (zero or more applications of $\forall$-L) results in a clause already present in $\mathcal{P}$ and not explicitly listed in the lower sequent. We have used EG (existential generalization) simply to identify existential variables with their appropriate instantiation — an artifact of our delayed instantiation during goal-oriented (backward) proof development. We use the inference figure BC (backchain), suggested in the previous section, to save a step and simplify our deduction. The inference figure BC replaces $\supset$-L and eliminates an initial upper sequent. (Also logic variables may be instantiated in the remaining upper sequent.) For example, the lowest use of BC in the deduction replaces the inference figure occurrence

$$\frac{\mathcal{P} \longrightarrow f(Y,X) \supset n(X) \qquad\qquad \mathcal{P}, b(Y) \longrightarrow b(Y)}{\mathcal{P}, [f(V,X) \supset n(X)] \supset b(V) \longrightarrow b(Y)} \supset\text{-L}$$

where $\supset$-L here is augmented by the instantiation of the logic variable V by Y.

With the modified inference figures suggested by the abstract interpreter of the preceding section, we observe that the modified **I**-proof is almost identical to that given in the QN-Prolog presentation.

Example: (A uniform proof of the Gabbay-Reyle example.)
As already noted, $\mathcal{P}$ corresponds here to the (universal closure of the) program presented earlier.

$$
\begin{array}{cl}
\mathcal{P}, f(Y,m), f(Z,m), cr(Z,m), c(s,m) \;\longrightarrow\; cr(s,m) & \\ \hline
\mathcal{P}, f(Y,m), f(Z,m), cr(Z,m), c(s,W) \;\longrightarrow\; cr(s,m) & \text{EG} \\ \hline
\mathcal{P}, f(Y,m), f(Z,m), cr(Z,m) \;\longrightarrow\; cr(s,m) & \text{CI} \\ \hline
\mathcal{P}, f(Y,X), f(Z,X), cr(Z,X), cr(s,m) \supset d(m) \;\longrightarrow\; d(X) & \text{BC} \\ \hline
\mathcal{P}, f(Y,X), f(Z,X), cr(Z,X) \;\longrightarrow\; d(X) & \text{CI} \\ \hline
\mathcal{P}, f(Y,X), f(Z,X) \wedge cr(Z,X) \;\longrightarrow\; d(X) & \wedge\text{-L} \\ \hline
\mathcal{P}, f(Y,X) \;\longrightarrow\; f(Z,X) \wedge cr(Z,X) \supset d(X) & \supset\text{-R} \\ \hline
\mathcal{P}, f(Y,X), [f(Z,W) \wedge cr(Z,W) \supset d(W)] \supset n(W) \;\longrightarrow\; n(X) & \text{BC} \\ \hline
\mathcal{P}, f(Y,X) \;\longrightarrow\; n(X) & \text{CI} \\ \hline
\mathcal{P} \;\longrightarrow\; f(Y,X) \supset n(X) & \supset\text{-R} \\ \hline
\mathcal{P}, [f(V,X) \supset n(X)] \supset b(V) \;\longrightarrow\; b(Y) & \text{BC} \\ \hline
\mathcal{P} \;\longrightarrow\; b(Y) & \text{CI} \\ \hline
\mathcal{P} \;\longrightarrow\; \exists Y b(Y) & \exists\text{-R}
\end{array}
$$

The deductive power of N-Prolog can be extended to a proof procedure for classical logic by the introduction of a new rule of inference called a restart rule. Gabbay [Gab85] calls the resulting system *NR-Prolog*. The propositional form of the restart rule states that at any point where the current goal is atomic, this atom can be replaced by the original query, and if this original query occurrence succeeds then the replaced goal is deemed to succeed. The original query might succeed at this point but not at the original occurrence because now the program has been enlarged by use of the rule for implications. One should note that any ancestor goal can replace a current goal because the ancestor goal could be regarded as the original query. (Use of clauses augmenting the program but introduced before the ancestor goal is not a problem, as we note below.)

The natural example to illustrate the use of the restart rule is Pierce's law. This formula is not valid intuitionistically, thus not provable in N-Prolog, as the reader can verify. We show that Pierce's law succeeds in NR-Prolog.
Example: (Pierce's law)

1.	$?\ ((a \supset b) \supset a) \supset a$	
2.	$(a \supset b) \supset a\ ?\ a$	rule for implication
3.	$(a \supset b) \supset a\ ?\ a \supset b$	using the only clause
4.	$(a \supset b) \supset a, a\ ?\ b$	rule for implication

5. $(a \supset b) \supset a, a \ ? \ (a \supset b) \supset a) \supset a$ restart
6. $(a \supset b) \supset a, a, (a \supset b) \supset a \ ? \ a$ rule for implication
7. success

As can be seen above, the restart subdeduction succeeds. Thus b succeeds from $\{(a \supset b) \supset a, a\}$. Thus the original query succeeds.

In the above example ancestor goal a could have been the restart goal. Although $(a \supset b) \supset a$ was added to the program before a became a goal, it is re-derived in the restart, which can always reproduce the path to a. Thus, any ancestor goal restart is always permissible in NR-Prolog.

The restart rule at the first-order level is a simple generalization of the propositional case. The restart rule for *QNR-Prolog*, the extension of QN-Prolog by addition of this rule, is precisely as for the propositional case except that the reintroduced original goal must have all its (existential) variables replaced by new existential variables. See [Gab85] for details.

The power of the restart rule is seen by comparing the following soundness and completeness theorem with the earlier theorem for QN-Prolog:

Proposition 6 *P ? Q succeeds in QNR-Prolog if*

$$\vdash \exists \vec{X}[(\forall \vec{U} P \supset Q]$$

holds in classical logic for program (database) P and query Q, where $\vec{X}$ denotes all the existential variables in P and Q and $\vec{U}$ denotes all the universal variables in P.

3.2 Near-Horn Prolog

We now shift our attention from an extension of Prolog (SLD-resolution) by hypothetical implication to an extension by disjunctive consequent. We wish to handle clauses with an indefinite conclusion, such as "jump_in_lake $\supset$ sink $\vee$ swim". This extension has received considerable attention recently, identified as a subarea (called Disjunctive Logic Programming) with a first book devoted to the subarea (see [LMR92]). Our focus is on proof procedures so we restrict ourselves to presenting (one version of) the Near-Horn Prolog (nH-Prolog) procedure mentioned earlier. This version of nH-Prolog was developed by Reed and Loveland [Ree88, Ree91, LR91] based on earlier versions devised by Loveland [Lov87, Lov91].

The astute reader may recognize that the introduction of non-Horn clauses such as above returns us to the full descriptive power of first-order logic since every clause in a conjunctive normal form (cnf) presentation of a formula can be written as an implication of atoms. Namely, negative literals of the cnf clause appear (positively) in the antecedent of the implication and positive literals of the cnf clause appear in the consequent. Since SLD-resolution is a restriction of SL-resolution (or ME), the appropriate procedure family seems at hand. Unfortunately, these procedures have properties not compatible with the character of logic programming. A severe problem with the use of ME-based procedures is the need to access any literal of the input clause in executing the extension operation. With the clauses in positive implication format this means accessing antecedent atoms as well as consequent atoms. This requires a notion of contrapositive implications, which reverses the direction of implication. This can strongly disrupt the user's sense of computation sequence, which is key to severely limiting the branching that occurs in Prolog execution. Moreover, the ability to catch errors in the formulation of the program is strongly dependent on the procedural reading capability of the programmer, and the disruption caused by the introduction of unintuitive implications is high. This seems a fatal flaw to the notion of "program" in comparison to "axiom clause set" that separates logic programming from automated theorem proving. There certainly is reason to look to preserve implication direction as SLD-resolution does.

There are several important properties enjoyed by SLD-resolution, and its implementation as Prolog, that one would seek to preserve in extending to a new domain. Prolog (SLD-resolution) is goal oriented, allows a declarative and procedural reading capability, has a linear input format, uses a positive implication format, and needs no contrapositive occurrences of the implications. Moreover, the process is intuitionistically sound, which insures a certain constructive nature to the inference structure. The attractiveness of nH-Prolog is that it shares these properties with Prolog, except that the linearity and the procedural reading properties are local only. (A certain type of contrapositive is used, but not one that changes the direction of implication.) The degree of locality depends on the number of uses of non-Horn clauses in the computation. Low use of non-Horn clauses yields more traditional computation sequences. This is a major reason why near-Horn Prolog is so-named; the intended domain is clause sets with few non-Horn clauses.

Like SLD-resolution, nH-Prolog is intuitionistically sound, although the process of preparation of clauses uses transformations that are classically but not intuitionistically provable. We will consider a version of nH-Prolog that meets the uniform proof criteria, somewhat unexpectedly given that the problem domain is all of first-order logic. (However, the abovementioned clause preparation accounts for some of the apparent gap in these two statements. For example, the formula $\sim\sim A \supset A$, not intuitionistically provable, simply transforms to query A and program A.)

The nH-Prolog procedure (in all variants) can be regarded as deduction by case analysis. A deduction is divided into *blocks*, where each block is essentially a Prolog deduction of one case. Information on deferred cases is retained in a "deferred head" list within each deduction line. The case-analysis approach is effective when the number of non-Horn clauses is low, but is not as effective as, say, ME-type procedures when many non-Horn clauses are present, such as in some logic puzzles and mathematical theorems.

We present a variant known as *Inheritance near-Horn Prolog*, or *InH-Prolog*, because it is relatively simple to describe, has an underlying complete first-order proof procedure, yet also seems to be the preferred version for implementation when building a compiler for the language. See [LR91] for a full discussion of related issues. See [RL92] for one comparison of this variant with N-Prolog and a Simplified Problem Reduction Format of Plaisted. After presenting the procedure we give two example deductions; these illustrate the procedure and also the role of disjunction in a database.

We consider the language format. The general clause form is

$$A_1; \ldots ; A_n \texttt{ :- } B_1, \ldots , B_m$$

where the A_i, B_j are atoms. The semicolons denote "or", and, as usual, the commas denote "and". Thus, the semantics for the above clause is the implication

$$B_1 \wedge B_2 \ldots \wedge B_m \supset A_1 \vee \ldots \vee A_n.$$

A definite Horn clause is given by $n = 1$. A (possibly disjunctive) fact is given by $m = 0$. The $n = 0$ case is not permitted as such, but is transformed into a special-case Horn clause by use of the special atom `FALSE`, which is given the semantics of "false" by its role in the InH-Prolog procedure. (As we shall see later, this atom corresponds to the logical symbol $\perp$ introduced in Subsection 2.2.) Thus query clause `?-` B_1, ... , B_m is written `FALSE :-` B_1, ... ,B_m and we use a fixed query `?- FALSE`.

The reason for the introduction of special atom `FALSE` involves the manner of handling negative clauses. Negative clauses may appear in the program itself, unlike Horn clause programs. A traditional query form, e.g. `?-` $q(X)$, is usable but then every negative program clause would have form $q(X)$ `:-` B_1, ... , B_m instead of `FALSE :-` B_1, ... , B_m. This makes the program dependent on the query, a very awkward trait. Answer retrieval is accomplished from the query clause `FALSE :-` Q, where Q is the user's query. Although formally a program clause, this (single) clause would be outside the compiled program and instead interpreted since it changes with each query. See [LR91] and [SL88] for further discussion.

Some terminology is needed before presenting the InH-Prolog procedure. The head of a clause now possibly has multiple atoms, called *head* atoms, sometimes just *heads* when context permits. (The term *body atom*, for atoms in the clause body, is available but rarely used.) Goal clauses, used in recording the deduction, are notated `:-` G_1, ..., G_n except for the initial goal clause, the query, which is notated `?-` G, in our case `?- FALSE`.

The summary we give applies to all the nH-Prolog variants except where noted.

An InH-Prolog deduction may be regarded as blocks of Prolog deductions where each block differs only slightly from standard Prolog. The basic Prolog reduction inference step is the primary inference step of the InH-Prolog block, with the adjustment that if the called program clause is non-Horn then the head atoms not called are put aside for use in another block. We now amplify the form of the deduction.

For the presentation of an InH-Prolog deduction we use a line format that displays the current goal clause, the active head list and the deferred head list. (The various head lists are defined below.) The format is:

goal_clause # active_head-list [deferred_head_list]

We first state the inference rules of InH-Prolog for a block.

Reduction — if G_s is the selected goal atom of the preceding goal clause G (in usual implementations this will always be the leftmost goal atom) and $H_1;\ldots;H_n$ `:-` $G_1,\ldots,G_m$ is a program clause, where G_s and H_i are unifiable, for some i, $1 \leq i \leq n$, then $(G_1, \ldots, G_m)\theta$ replaces $G_s\theta$ in $G\theta$, where θ is the most general unifier of G_s and H. The $H_i\theta$, $j \neq i$, are added to the *deferred head* list, with no variable renaming. Both the active and deferred head list pass to the next line; changes from line to line include only possible instantiation of variables shared with the goal clause and additions to the deferred head list as just described. The deferred head list is placed between brackets.

Cancellation — if G_s is the selected goal in goal clause G and G_s unifies with an active head A (defined below), then the next goal clause is $(G - \{G_s\})\theta$, where θ is the unifier of G_s and A.

Variables in head list atoms can be instantiated during reduction through shared variables with goal clause atoms or during cancellation.

We now consider block structure, which is reflected in the initialization of the block. Blocks serve as subdeductions, a single case consideration in the analysis-by-cases characterization of the InH-procedure.

Block initiation — the first line of a block has the query `?- FALSE` as goal clause. For *restart* blocks (blocks other than the first, or *start*, block) lists of *active* head atoms and deferred head atoms are declared that accompanies each line of the block. If a block terminates successfully by deriving the empty clause as for Prolog, but with a nonempty deferred head list, then a successor restart block is created. A *distinguished* active head is defined for the new block, chosen by a user supplied selection function from the deferred head list of the final line of the preceding block. (For a standard implementation it will be the leftmost deferred head.) The deferred head list is the deferred head list of the final line of the preceding block, minus the now distinguished active head. For other nH-Prologs this defines the active and deferred head lists of the new block but InH-Prolog has also as active heads the active heads of the block in which the new distinguished active head was deferred. If one represents the block structure in tree form with a child block for each deferred head in a block, then the active heads in the new child block include the parent block deferred head as distinguished active head and the active heads of the parent block as the other active heads. No deferred head list exists, as the heads are immediately recorded in the children blocks. (The possibility that multiple heads from a clause can share variables complicates the tree format by requiring that binding information be passed between blocks. In the sequential format we are outlining, acquiring heads from the previous block handles shared bindings. However, special recording mechanisms are needed in InH-Prolog to identify the correct active heads in the preceding block.)

In the start block where no active head list exists, the separator symbol # is omitted until deferred heads are introduced.

The deduction terminates successfully when no new block can be formed, which occurs when the deferred head list is empty.

We distinguish the newest active head primarily because of the key cancellation requirement discussed in the next paragraph, but also because nH-Prolog deductions must have one restart block per deferred head, and the distinguished active head is the appropriate head atom to identify with the block. The active heads serve as conditional facts in the case-analysis view of the nH-Prologs.

The InH-Prolog procedure has added requirements that pertain to search. A non-obvious property whose absence would make the procedure impractical is the cancellation requirement. This requirement states that a restart block is to be accepted in a deduction only if a cancellation has occurred using the distinguished active head. (Cancellation with other active heads may occur also, of course, but do not satisfy the requirement.) This demand keeps blocks relevant, and prunes away many false search branches. The underlying proof procedure is a complete proof procedure with this feature. We return to the importance of this later.

Another important search feature is *progressive search*. Although too complex to treat fully here we outline the feature. Recall that a restart block in InH-Prolog always begins with the initial query `FALSE`. (This is not true of the Unit nH-Prolog variant, originally called Progressive nH-Prolog [Lov87, Lov91] because it first incorporated this important search feature. However, this feature is useful for all variants.) If the program clauses were processed exactly as its "parent" block, then it would duplicate the computation with no progress made, unless the new distinguished active head can invoke a cancellation. Progressive search starts the search of the restart block where the "parent" block finished (roughly), so that a new search space is swept. More precisely, the initial portion of the restart block deduction duplicates the initial portion of the block where the current distinguished active head was deferred, the initial portion ending at that point of deferral. (The path to this point can be copied with no search time accrued.) In effect we insert a FAIL at this point and let normal Prolog backtracking advance the search through the remainder of the search space. If this search fails then the search continues "from the top", as a regular search pattern would begin the block. The search halts in failure when the initial path to the deferred head point is again reached. For a more precise description see [Lov91]. An interpreter for Unit nH-Prolog has proven the worth of this search strategy.

We present two examples of InH-Prolog deductions. The first has very short blocks but has multiple non-Horn clauses and an indefinite answer. The second example has less trivial blocks and yields a definite answer.

An answer is the disjunction of the different instantiations of the query clause which begin InH-Prolog blocks. Not every block begins with a query clause use; negative clauses also have head atom `FALSE` and may begin blocks. However, negative clauses are from the program so represent known information. Only uses of the query clause yield new information and warrant inclusion in the answer to a query. Nor does each use of a query clause yield a different instantiation; identical instantiations of answer literals are merged.

The deduction below shows a successful computation without any indication of the search needed to find it, as is true of any displayed deduction. It is natural to wonder how much search occurs before the query "men_req(X, Cond)" calls the correct clause in the successive restart blocks. (Following Prolog, strings beginning with capital letters are variables.) At the completion of the deduction we show that the cancellation requirement kills the attempt to repeat the start block and thus forces the system to backtrack immediately upon block failure to move to the next (and correct) clause to match the query.

Example: (The manpower requirement example)

Program:

```
men_req(8,normal) :- cond(normal).
men_req(20,wind_storm) :- cond(wind_storm).
men_req(30,snow_removal) :- cond(snow_removal).
cond(snow_removal); cond(wind_storm) :- cond(abnormal).
cond(normal) ; cond(abnormal).
```

Query:

```
?- men_req(X,Cond).
```

Query Clause:

```
FALSE :- men_req(X,Cond).
```

Deduction:

```
    ?- FALSE
    :- men_req(X,Cond)
    :- cond(normal)              % X=8, Cond=normal
    :-                         # [cond(abnormal)]
restart
```

```
        ?- FALSE              # cond(abnormal)
        :- men_req(X1,Cond1)  # cond(abnormal)
        :- cond(wind_storm)   # cond(abnormal)
                                 % X=20, Cond1=wind_storm
        :- cond(abnormal)     # cond(abnormal) [cond(snow_removal)]
        :-                    # cond(abnormal) [cond(snow_removal)]
                                 % cancellation with distinguished head
restart
        ?- FALSE              # cond(snow_removal),cond(abnormal)
                                 % The new distinguished head is placed
                                 % leftmost in the active head list
        :- men_req(X2,Cond2)  # cond(snow_removal),cond(abnormal)
        :- cond(snow_removal) # cond(snow_removal),cond(abnormal)
                                 % X=30, Cond2=snow_removal
        :-                    # cond(snow_removal),cond(abnormal)
                                 % cancellation with distinguished head
```

The query answer is the disjunction of the various instantiations of the user query for successful blocks. Here the answer is:

men_req(8,normal) OR men_req(20,wind_storm) OR men_req(30,snow-removal)

We now show that the cancellation requirement aids the search process in the manner discussed before the deduction. A restart block with standard search rules would recompute the computation of the start block. However, if the following restart block were to follow the start block, and not be eliminated, then an infinite loop would be entered. (This restart block essentially repeats the start block).

```
restart
        ?- FALSE              # cond(abnormal)
        :- men_req(X1,Cond1)  # cond(abnormal)
        :- cond(normal)       # cond(abnormal)
        :-                    # cond(abnormal) [cond(abnormal)]
```

Because the distinguished active head "cond(abnormal)" is not used in cancellation this block would not be retained, normal backtracking would occur and the block of the deduction would be obtained. The progressive search feature discussed previously would also keep this block from being computed directly after the start block; the cancellation requirement allows a more standard search order to be used if desired. Also this illustrates the cancellation requirement mechanism; this discard feature is used many times other than repressing near-duplicate blocks.

The second example yields a definite answer. This is a commuting problem with an alternate behavior depending on the status of the office parking lot. If the problem were formulated to return a plan, then an indefinite answer would be returned [LR91]. This example illustrates what we believe is a common structure for disjunctive logic programs. The non-Horn clause records the expected case ("park in office lot") and the error case ("lot is full"). Recovery possibilities are confirmed by the completion of the computation.

Example: (The commute example.)

Program:

```
car(al).
drives(X,office_bldg) :- car(X).
gets_to(X,office_bldg) :- parks(X,office_bldg).
goes_to_mtg(X) :- gets_to(X,office_bldg).
parks(X,commercial) :- car(X),lot_full(office_bldg).
finds(X,office_bldg) :- drives(X,office_bldg), parks(X,commercial).
gets_to(X,office_bldg) :- finds(X,office_bldg).
parks(X,office_bldg); lot_full(office_bldg) :- drives(X,office_bldg).
```

Query:

```
?- goes_to_mtg(X).
```

Query Clause:

```
FALSE :- goes_to_mtg(X)
```

Deduction:

```
?- FALSE
:- goes_to_mtg(X)
:- gets_to(X,office_bldg)
:- parks(X,office_bldg)
:- drives(X,office_bldg)                # [lot_full(office_bldg)]
:- car(X)                               # [lot_full(office_bldg)]
:-                                      # [lot_full(office_bldg)]
                                            % X=al
```

restart

```
?- FALSE
:- goes_to_mtg(X1)                                 # lot_full(office_bldg)
:- gets_to(X1,office_bldg)                         # lot_full(office_bldg)
:- finds(X1,office_bldg)                           # lot_full(office_bldg)
:- drives(X1,office_bldg), parks(X1,commercial)    # lot_full(office_bldg)
:- car(X1), parks(X1,commercial)                   # lot_full(office_bldg)
:- parks(al,commercial)                            # lot_full(office_bldg)
                                                       % X1=al
:- car(al), lot_full(office_bldg)                  # lot_full(office_bldg)
:- lot_full(office_bldg)                           # lot_full(office_bldg)
                                                       % cancellation
:-                                                 # lot_full(office_bldg)
```

Query answer:

goes_to_mtg(al)

Observe that the query answer above is the merge of the disjunction of two identical atoms, each corresponding to a block.

Unit nH-Prolog differs from InH-Prolog primarily by the use of only one active head atom, the distinguished active head, and by a possibly different restart block initial goal clause. The use of only one active head means a "constant time" inner loop speed (when a no-occurs-check unification is used, as for Prolog) rather than an inner-loop speed that varies with the length of the active list to be checked. (Pre-processing by a compiler usually makes the possibly longer active head list of InH-Prolog also process in near-constant time. This is not always possible, such as when multiple entries to the active head list share the same predicate letter.) The different restart initial goals can allow a shorter deduction to be written (and yield a shorter search time) but represent another branch point in the search and may in fact lengthen search time. (The latter is much less costly than first considerations indicate.) The search time tradeoff here is hard to quantify. Evidence at present suggests that InH-Prolog is superior when compilation is possible. For interpreters, Unit nH-Prolog seems to have the edge on speed, but is somewhat more complex to code.

We now consider how to represent the underlying inference system of InH-Prolog in a uniform proof format. One way is to rewrite non-Horn clauses in a classically equivalent form so that only rules acceptable to the uniform proof concept are used. We illustrate this approach by example, writing clauses as traditional implications (not Prolog notation), using $\supset$ for implies and interpreting the atom `FALSE` as an alternative notation for $\perp$. Given the non-Horn clause $A \supset (B \vee C)$, we may rewrite this in the classically equivalent forms $(A \wedge (B \supset \texttt{FALSE})) \supset C$ and $(A \wedge (C \supset \texttt{FALSE})) \supset B$ with both clauses needed to replace $A \supset (B \vee C)$. With use of clauses of this form, InH-Prolog derivations can indeed be described as uniform proofs.

We choose a different approach for our major illustration, one that does not require us to rewrite the non-Horn clauses. We obtain the effect of case analysis and block initiation used by InH-Prolog by use of a derived rule meeting the uniform proof conditions. We justify the derived *restart* inference figure

$$
\begin{prooftree}
\AxiomC{$\mathcal{P}, A \longrightarrow \texttt{FALSE}$}
\AxiomC{$\mathcal{P}, B \longrightarrow B$}
\RightLabel{restart}
\BinaryInfC{$\mathcal{P}, A \vee B \longrightarrow B$}
\end{prooftree}
$$

by the following derivation:

$$
\begin{prooftree}
\AxiomC{$\mathcal{P}, A \longrightarrow \texttt{FALSE}$}
\RightLabel{$\perp$-R}
\UnaryInfC{$\mathcal{P}, A \longrightarrow B$}
\AxiomC{$\mathcal{P}, B \longrightarrow B$}
\RightLabel{$\vee$-L.}
\BinaryInfC{$\mathcal{P}, A \vee B \longrightarrow B$}
\end{prooftree}
$$

There is a symmetric rule for $\mathcal{P}, A \vee B \longrightarrow A$ and this obviously generalizes for different size disjunctive clauses. The similarity in effect between the restart rule and the rewriting of the non-Horn clauses suggested earlier is obvious.

We give a uniform proof for the manpower requirement example. To shorten the proof tree we will combine the CI (Clause Incorporation) and BC (backchain) into one inference figure. Single symbol abbreviations are used for predicate and constant names and $\mathcal{P}$ denotes the formulas constituting the program for the example. We again use "existential" variables in the CI inference figure to signify the creation of one instance of a clause when the proof is read from the bottom upward.

$$
\begin{prooftree}
\AxiomC{$\mathcal{P}, c(a) \longrightarrow c(a)$}
\AxiomC{$\mathcal{P}, c(a), c(s) \longrightarrow c(s)$}
\RightLabel{BC}
\UnaryInfC{$\mathcal{P}, c(a), c(s), c(s) \supset m(30, s) \longrightarrow m(X_2, Y_2)$}
\RightLabel{CI, BC}
\UnaryInfC{$\mathcal{P}, m(X_2, Y_2) \supset \texttt{FALSE} \longrightarrow \texttt{FALSE}$}
\RightLabel{CI}
\UnaryInfC{$\mathcal{P}, c(a), c(s) \longrightarrow \texttt{FALSE}$}
\AxiomC{$\mathcal{P}, c(a), c(w) \longrightarrow c(w)$}
\RightLabel{restart}
\BinaryInfC{$\mathcal{P}, c(a), c(s) \vee c(w) \longrightarrow c(w)$}
\RightLabel{$\supset$-L}
\BinaryInfC{$\mathcal{P}, c(a), c(a) \supset c(s) \vee c(w) \longrightarrow c(w)$}
\RightLabel{CI, BC}
\UnaryInfC{$\mathcal{P}, c(a), c(w) \supset m(20, w) \longrightarrow m(X_1, Y_1)$}
\RightLabel{CI, BC}
\UnaryInfC{$\mathcal{P}, c(a), m(X_1, Y_1) \supset \texttt{FALSE} \longrightarrow \texttt{FALSE}$}
\RightLabel{CI}
\UnaryInfC{$\mathcal{P}, c(a) \longrightarrow \texttt{FALSE}$}
\AxiomC{$\mathcal{P}, c(n) \longrightarrow c(n)$}
\RightLabel{restart}
\BinaryInfC{$\mathcal{P}, c(n) \vee c(a) \longrightarrow c(n)$}
\RightLabel{CI, BC}
\UnaryInfC{$\mathcal{P}, c(n) \supset m(8, n) \longrightarrow m(X, Y)$}
\RightLabel{CI, BC}
\UnaryInfC{$\mathcal{P}, m(X, Y) \supset \texttt{FALSE} \longrightarrow \texttt{FALSE}$}
\RightLabel{CI}
\UnaryInfC{$\mathcal{P} \longrightarrow \texttt{FALSE}$}
\end{prooftree}
$$

We conclude this section with another example, one where the capability of the InH-Prolog procedure might seem to extend beyond the ability to represent the computation by a uniform proof. Indeed, the ability to model the computation by a uniform proof depends on the variant of InH-Prolog used, because the role of the query clause is significant in the associated uniform proof.

We are given the program consisting of the single clause $p(a); p(b)$, and ask the query

?- $p(X)$.

The sequent $p(a) \vee p(b) \longrightarrow \exists X p(X)$ has no uniform proof, so it is instructive to see the uniform proof for this InH-Prolog computation. By convention, the query is replaced by query clause `FALSE :-` $p(X)$ and query ?- `FALSE`. This transformation is justifiable in classical logic; in particular, the sequent $(\forall X (p(X) \supset \mathtt{FALSE})) \supset \mathtt{FALSE} \longrightarrow \exists X p(X)$ has a **C**-proof. The InH-Prolog computation of query ?- `FALSE` given program $p(a) \vee p(b), p(X) \supset \mathtt{FALSE}$ succeeds, giving answer $p(a) \vee p(b)$. Using the various conventions described earlier and denoting the set $\{\forall X (p(X) \supset \mathtt{FALSE}), p(a) \vee p(b)\}$ by $\mathcal{P}$, the associated uniform proof is given below.

$$
\begin{prooftree}
\AxiomC{$\mathcal{P}, p(a) \longrightarrow p(a)$}
\RightLabel{EG}
\UnaryInfC{$\mathcal{P}, p(a) \longrightarrow p(Z)$}
\AxiomC{$\mathcal{P}, p(a), \mathtt{FALSE} \longrightarrow \mathtt{FALSE}$}
\RightLabel{$\supset$-L}
\BinaryInfC{$\mathcal{P}, p(a), p(Z) \supset \mathtt{FALSE} \longrightarrow \mathtt{FALSE}$}
\RightLabel{CI}
\UnaryInfC{$\mathcal{P}, p(a) \longrightarrow \mathtt{FALSE}$}
\AxiomC{$\mathcal{P}, p(b) \longrightarrow p(b)$}
\RightLabel{EG}
\UnaryInfC{$\mathcal{P}, p(b) \longrightarrow p(Y)$}
\RightLabel{restart}
\BinaryInfC{$\mathcal{P}, p(a) \vee p(b) \longrightarrow p(Y)$}
\RightLabel{CI}
\UnaryInfC{$\mathcal{P} \longrightarrow p(Y)$}
\AxiomC{$\mathcal{P}, \mathtt{FALSE} \longrightarrow \mathtt{FALSE}$}
\RightLabel{$\supset$-L}
\BinaryInfC{$\mathcal{P}, p(Y) \supset \mathtt{FALSE} \longrightarrow \mathtt{FALSE}$}
\RightLabel{CI}
\UnaryInfC{$\mathcal{P} \longrightarrow \mathtt{FALSE}$}
\end{prooftree}
$$

4 Conclusion

The material presented in this paper may be read in two ways. The first way is to consider the procedures as primary, starting with the historical introduction of procedures. This leads to the SLD-resolution procedure, which is the inference foundation for Prolog and anchors the traditional view of logic programming. Two extensions of the traditional notion of logic programming also appear. Included is an abstraction of the concept of logic programming that may be regarded as a derivative attempt to encode the essence of these procedures. An alternative view is to take the abstraction as primary and as setting the standard for acceptable logic programming languages. If this viewpoint is adopted, the abstraction provides guidance in constructing goal-directed proof procedures for logic programming — such procedures will essentially be ones that search for uniform proofs in the relevant context. Horn clause logic and the associated SLD-resolution procedure are illustrations of this viewpoint. However, richer exploitations of the general framework are also possible and this issue is discussed at length. The two procedures that conclude the paper serve as additional novel illustrations of the notion of searching for a uniform proof. N-Prolog is an excellent example of an extension of SLD-resolution that fits the uniform proof structure. The second example, near-Horn Prolog also fits the uniform proof structure when we specify carefully the version (InH-Prolog) and the input format (use of query clause). That InH-Prolog is in fact a proof procedure for full (classical) first-order logic makes the fit within a uniform proof structure particularly interesting. However, recall that conversion of a given valid formula into the positive implication refutation logic format used by InH-Prolog may utilize several equivalences valid only classically.

Although the second way of reading the paper most likely is not the viewpoint the reader first adopts, it is a viewpoint to seriously consider. If the reader agrees with the paradigm captured by the structure of uniform proof, then he/she has a framework to judge not just the procedures presented here but future procedures that seek to extend our capabilities within the logic programming arena.

References

[AK90] H. Aït-Kaci. *Warren's Abstract Machine: a Tutorial Reconstruction.* MIT Press, Cambridge, MA, 1990.

[AL91] O.L. Astrachan and D.W. Loveland. METEORs: High performance theorem provers for Model Elimination. In R.S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.

[And76] P. Andrews. Refutations by Matings. *IEEE Transactions on Computers*, C-25:801–807, 1976.

[AS92] O.L. Astrachan and M.E. Stickel. Caching and lemmaizing in Model Elimination theorem provers. In Bibel and Kowalski, editors, *Proceedings of the Eleventh Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence 607, pages 224–238. Springer-Verlag, Berlin, June 1992.

[BCLM91] S. Bose, E.M. Clarke, D.E. Long, and S. Michaylov. PARTHENON: a parallel theorem prover for non-Horn clauses. *Journal of Automated Reasoning*, pages 153–181, 1991.

[Bib82] W. Bibel. *Automated Theorem Proving.* Vieweg Verlag, Braunschweig, 1982.

[Bow82] K. A. Bowen. Programming with full first-order logic. In J. E. Hayes, D. Michie, and Y-H Pao, editors, *Machine Intelligence 10*, pages 421–440. Halsted Press, 1982.

[CL73] C. Chang and R.C. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, 1973.

[Fit69] Melvin Fitting. *Intuitionistic Logic, Model Theory and Forcing.* North-Holland Publishing Company, 1969.

[Fit85] Melvin Fitting. A Kripke-Kleene semantics for logic programming. *Journal of Logic Programming*, 2(4):295–312, 1985.

[Fit90] Melvin Fitting. *First-order Logic and Automated Theorem Proving.* Springer-Verlag, 1990.

[Gab85] D.M. Gabbay. N-Prolog: an extension of Prolog with hypothetical implication, Part 2. *Journal of Logic Programming*, 4:251–283, 1985.

[Gen69] Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North Holland Publishing Co., 1969.

[GR68] C. Green and B. Raphael. The use of theorem-proving techniques in systems. In *Proceedings of the 23rd National Conference of the Association of Computing Machinery*, pages 169–181, Princeton, 1968. Brandon Systems Press.

[GR84a] D. Gabbay and U. Reyle. N-Prolog: An extension to Prolog with hypothetical implications I. *Journal of Logic Programming*, 1(4):319–355, 1984.

[GR84b] D.M. Gabbay and U. Reyle. N-Prolog: an extension of Prolog with hypothetical implication, Part 1. *Journal of Logic Programming*, 4:319–355, 1984.

[GR87] J. Gallier and S. Raatz. Hornlog: A graph-based interpreter for general Horn clauses. *Journal of Logic Programming*, 4(2):119–156, 1987.

[Gre69] C. Green. Theorem-proving by resolution as a basis for questions-answering systems. In Meltzer and Michie, editors, *Machine Intelligence 4*, pages 183–205. Edinburgh University Press, Edinburgh, 1969.

[Har60] R. Harrop. Concerning formulas of the types $A \rightarrow B \vee C$, $A \rightarrow (Ex)B(x)$ in intuitionistic formal systems. *Journal of Symbolic Logic*, 25:27–32, 1960.

[Hil] Robert Hill. LUSH-resolution and its completeness. DCL Memo No. 78, School of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, August 1974.

[HM94] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, May 1994.

[HP91] James Harland and David Pym. The uniform proof-theoretic foundation of linear logic programming (extended abstract). In Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the 1991 International Logic Programming Symposium*, pages 304–318. MIT Press, 1991.

[KK71] R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, pages 227–260, 1971.

[Kow70] R. Kowalski. Search strategies for theorem proving. In Meltzer and Michie, editors, *Machine Intelligence 6*, pages 181–201. Edinburgh University Press, Edinburgh, 1970.

[Kow79] Robert Kowalski. *Logic for Problem Solving*. North-Holland, 1979.

[LMR92] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunction Logic Programming*. MIT Press, Cambridge, MA, 1992.

[Lov68] D.W. Loveland. Mechanical theorem proving by model elimination. *Journal of the Association for Computing Machines*, pages 236–251, 1968.

[Lov69] D.W. Loveland. A simplified format for the model elimination procedure. *Journal of the Association for Computing Machines*, pages 349–363, 1969.

[Lov70] D.W. Loveland. A linear format for resolution. In *Symposium on Automatic Demonstration*, Lecture Notes in Mathematics 125, pages 147–162. Springer-Verlag, Berlin, 1970.

[Lov78] D.W. Loveland. *Automated Theorem Proving: a Logical Basis*. North-Holland, Amsterdam, 1978.

[Lov87] D.W. Loveland. Near-Horn Prolog. In *Proceedings of the Fourth International Conference and Symposium on Logic Programming*, pages 456–469. MIT Press, 1987.

[Lov91] D.W. Loveland. Near-Horn Prolog and beyond. *Journal of Automated Reasoning*, pages 1–26, 1991.

[LR91] D.W. Loveland and D.W. Reed. A near-Horn Prolog for compilation. In Lassez and Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 542–564. MIT Press, Cambridge, MA, 1991.

[LSBB91] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: a high-performance theorem prover. *Journal of Automated Reasoning*, pages 183–212, 1991.

[LT84] J. Lloyd and R. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[Luc70] D. Luckham. Refinement theorems in resolution theory. In *Symposium on Automatic Demonstration*, Lecture Notes in Mathematics 125, pages 163–190. Springer-Verlag, Berlin, 1970.

[McC88a] L. Thorne McCarty. Clausal intuitionistic logic I. Fixed point semantics. *Journal of Logic Programming*, 5(1):1–31, 1988.

[McC88b] L. Thorne McCarty. Clausal intuitionistic logic II. Tableau proof procedures. *Journal of Logic Programming*, 5(2):93–132, 1988.

[Mil89a] Dale Miller. Lexical scoping as universal quantification. In G. Levi and M. Martelli, editors, *Sixth International Logic Programming Conference*, pages 268–283, Lisbon, Portugal, June 1989. MIT Press.

[Mil89b] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[Mil92] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Nad93] Gopalan Nadathur. A proof procedure for the logic of hereditary Harrop formulas. *Journal of Automated Reasoning*, 11(1):115–145, August 1993.

[NJK93] Gopalan Nadathur, Bharat Jayaraman, and Keehang Kwon. Scoping constructs in logic programming: Implementation problems and their solution. Technical Report CS-1993-17, Department of Computer Science, Duke University, July 1993.

[NM88] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[Pau87] Lawrence R. Paulson. The representation of logics in higher-order logic. Technical Report Number 113, University of Cambridge, Computer Laboratory, August 1987.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE Computer Society Press, June 1989.

[Poh71] I. Pohl. Bi-directional search. In Meltzer and Michie, editors, *Machine Intelligence 6*, pages 127–140. Edinburgh University Press, Edinburgh, 1971.

[Pra65] Dag Prawitz. *Natural Deduction: A Proof-Theoretical Study.* Almqvist & Wiksell, 1965.

[Ree88] D.W. Reed. Near-Horn Prolog: A first-order extension to Prolog. Master's thesis, Duke University, May 1988.

[Ree91] D.W. Reed. *A Case-analysis Approach to Disjunctive Logic Programming.* PhD thesis, Duke University, December 1991.

[RL92] D.W. Reed and D.W. Loveland. A comparison of three Prolog extensions. *Journal of Logic Programming*, 12:25–50, 1992.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machines*, pages 23–41, 1965.

[SL88] B.T. Smith and D.W. Loveland. A simple near-Horn Prolog interpreter. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Seattle, 1988.

[SL90] J. Schumann and R. Letz. PARTHEO: a high performance parallel theorem prover. In M. Stickel, editor, *Proceedings of the Tenth International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence 449, pages 40–56, Berlin, June 1990. Springer-Verlag.

[Sti84] M.E. Stickel. A Prolog Technology Theorem Prover. *New Generation Computing*, pages 371–383, 1984.

[Sti88] M.E. Stickel. A Prolog Technology Theorem Prover: implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, pages 353–380, 1988.

[Tro73] A. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis.* Number 344 in Lecture Notes in Mathematics. Springer-Verlag, 1973.

[War] D.H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, October 1983.

[WCR64] L. Wos, D. Carson, and G. Robinson. The unit preference strategy in theorem proving. In *AFIPS Conference Proceedings 26*, pages 615–621, Washington, D.C., 1964. Spartan Books.

[WOL91] L. Wos, R. Overbeek, and E. Lusk. Subsumption, a sometimes undervalued procedure. In Lassez and Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 3–40. MIT Press, Cambridge, MA, 1991.

[WRC65] L. Wos, G. Robinson, and D. Carson. Efficiency and completeness of the set of support strategy in theorem proving. *Journal of the Association for Computing Machines*, pages 536–541, 1965.