Proof Theory and Semantics of Logic Programs

Haim Gaifman and Ehud Shapiro

Abstract

We develop a resolution logic that is based on direct proofs rather than on proofs by refutations. The deductive system studied has clauses as its formulas and resolution as the sole inference rule.

We analyze this deductive system using a novel representation of resolution proofs, called resolution graphs, and obtain a general completeness theorem: a clause is a logical consequence of a set of clauses iff it is either tautological or is subsumed by a clause derivable from that set. All previous completeness results for resolution and its variants concerned either refutations or the derivability of atoms. Our completeness theorem generalizes these results (including strong completeness of SLD resolution) to clauses.

Proof-theoretic distinctions in resolution logic are much finer than distinctions based on logical equivalence. For example, from a contradiction (the empty clause) nothing but the empty clause itself is derivable; a clause derivable from a set of clauses must be in the vocabulary of that set, thus generally not all tautologies are derivable.

In a previous paper [GaSh] we provided a framework for studying the relations between observables, compositions, and fully abstract semantics. Within this framework we developed a model-theoretic compositional semantics for logic programs, and investigated the fully abstract equivalences induced by various notions of composition. This paper continues that study using the proof theory of resolution logic. This proof theory gives rise to various semantics for logic programs that reflect more operational details than does the model-theoretic semantics. The basic one is the answer substitution semantics [FLMP] which distinguishes between logically equivalent programs that may give different answer substitutions to the same goal. We show that taking answer substitutions as observables is equivalent to taking the set of derivable atoms, extend this basic semantics to various forms of program composition, and obtain fully abstract invariants for these compositions.

1 Introduction

The resolution principle [Ro], which was at the foundation of most research in automatic theorem proving and is at the basis of logic programming, was introduced as a mechanism for constructing indirect proofs via refutations. As a result its associated proof theory was rather weak, since it considered only the eventual derivation of a contradiction from a set of inconsistent premises, rather than the set of statements derivable from any set of premises. Hence the first, and to date the major completeness result for resolution, due to Robinson [Ro], states that if a set of clauses is unsatisfiable then repeated applications of resolution would eventually derive the empty clause.

Logic programming necessitated refinement of the resolution principle, in order to provide the "answer substitution", the output of the computation of a logic program (set of definite clauses) on a goal statement (a negative clause). A mechanism for extracting an answer substitution from a resolution proof of Horn (negative and definite) clauses was defined. The mechanism proposed, however, was roundabout, and was
specialized to a certain kind of resolution proofs, now called SLD resolution.

Additional completeness results were obtained for SLD resolution. Hill [Hi] showed that if a set of definite clauses and one negative clause is unsatisfiable, then SLD resolution would derive from this set the empty clause. Clark [Cl] showed that the set of answer substitutions computed by SLD resolution is complete, in that it provides all the most general instances of an atom that are a logical consequence of a program. His strong completeness theorem states that this set can be computed independently of the computation rule, i.e. the order in which body atoms are resolved.

In this paper we investigate Resolution Logic: a deductive system with no axioms that uses resolution to obtain direct proofs of clauses from sets of clauses. We analyze this deductive system using a novel representation of resolution proofs, called resolution graphs. In a resolution graph the layout of all clauses to be resolved is given ahead in the form of a graph, where nodes are labeled by clauses and where edges connect the occurrences of literals to be resolved against each other. To evaluate a resolution graph we carry out a sequence of "edge resolutions". The resolving of an edge results in the deletion of the edge and the replacement of the two clauses incident on the edge by their resolvent, where all other edges to literal occurrences in the graph remain intact.

For example, below is a resolution graph whose clauses are taken from the program:

\[ A \rightarrow R(x, y), S(z), S(y) \]
\[ R(fu, fu) \rightarrow S(fu) \]
\[ S(fu) \rightarrow T(v), B \]
\[ B \rightarrow T(c) \]
\[ T(fu) \rightarrow T(fz) \]

where \( A, B \) are 0-ary predicates, \( f \) is a function symbol, \( c \) is an individual constant and \( \bar{X} \) is used for the negation of \( X \).

Boxes are the nodes, the edges are \( e_1, e_2, e_3, e_4 \), each links two literal-occurrences which will be resolved with each other, e.g., \( e_1 \) links \( R(x, y) \) and \( R(fu, fu) \).
atom (or, in general, a conjunctive goal and a derived conjunction).

Our analysis clarifies a distinction which passed mostly unnoticed in studies of resolution: between clauses as sets of literals on one hand, and clauses as either multisets or sequences of literals on the other. In [Ro] they are sets and so, essentially, they are in [ChLe]. In [ApEm] and [Ll] they are sequences. The practice of logic programming treats clauses as sequences (e.g. Prolog) or multisets (e.g. concurrent logic languages) for obvious performance reasons. The difference between the two approaches lies in the effect of forming a resolvent. Applying an mgu and forming multiset unions may lead to repeated occurrences of the same literal. If clauses are sets only one occurrence remains in the resolvent. Such a pruning does not take place if the clauses are multisets or sequences. While the distinction does not matter in a semantics based on logical equivalence, it is crucial for the more refined proof-theoretic approach. There is a natural smooth proof theory for multisets or sequences, but not so for sets. We do, though, get the completeness results for both; for sets the proof is harder.

In a previous paper [GaSh] we proposed a framework for discussing fully abstract semantics, which exposes the interrelations between the choices of observables, compositions, and meanings. In this framework the choice of observables and compositions determines a unique coarsest equivalence between programs, called the fully abstract equivalence. A semantics is a mapping from programs to some mathematical objects, called program meanings. A semantics is fully abstract if it induces the fully abstract equivalence. Note that the equivalence class of the coarsest equivalence constitutes a trivial fully abstract semantics. The aim is, however, to get non trivial invariants that provide insights into the nature of the equivalence.

In [GaSh] the observables are defined model-theoretically as the logically implied atoms. Various compositions are considered. In the non-modular case, one simply considers unions of programs (which might share parts of their vocabulary). A more sophisticated notion concerns logic modules. A module is a logic program with a classification of its predicates into exported, imported, and internal. Exported predicates are those that the module exports to the environment (user, or other modules); imported ones are those it imports from the environment, they cannot occur in the heads of clauses; internal predicates are local to it and are hidden from the environment. A natural notion of module composition is defined. It turns out that all the results of the non-modular composition carry over to the modular case through relativization to I/E clauses, where an I/E clause is one in which all predicates in the body are imported and the head predicate is exported. The results of the model-theoretic studies are summarized in the top-half of the table below.

In the present work we use more distinguishing observables: atoms provable from the program. As noted above we show this to be equivalent to taking the answer substitution pairs as observables. With similar notions of composition the results are summarized in the bottom half of the table. The reason for the additional line compared to the top half is that, unlike in the model-theoretic semantics, unions with atoms and unions with ground atoms yield different semantics.

The novel notions in this table — body-instance, body-closure, body-minimal, and associated implication — are defined in Section 6.

This semantics is "intermediate" between the model theoretic semantics of logic programs, and the still more operational semantics of concurrent logic languages [GCLS].

The rest of the paper is organized as follows. Section 2 provides the basic concepts and definitions. Section 3 introduces resolution graphs and proves the order-independence theorem. Section 4 contains the completeness result. Section 5 studies the answer substitution semantics, and Section 6 the compositional semantics.

## 2 Basic Concepts

### Syntactic Objects, Substitutions and Unifiers.

A vocabulary consists of function symbols (including individual constants), predicates and individual variables. We assume that all vocabularies share the same infinite list of individual variables. Terms are constructed as usual from individual variables, constants and function symbols. Atomic formulas, atoms for short, are constructed as usual by applying predicates to terms.

A literal is an atom (positive literal) or a negated atom (negative literal). If \( L \) is a literal then \( \neg L \) is the negated literal (where a double negation is dropped: \( \neg \neg L = L \)).

We use "syntactic objects" in an intuitive sense which covers structures having terms as components. In particular, any sequence of syntactic objects is a syntactic object. \( vars(T) \) is the set of individual variables occurring in the syntactic object \( T \).

A substitution is a simultaneous assignment of terms to variables: \( v_1 := t_1, \ldots, v_n := t_n \) (written also as \( v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n \)) with \( v_i \) distinct and each \( v_i \) different from its corresponding \( t_i \). We say that the assignment binds the \( v_i \)’s. The substitution’s domain is
a substitution and \( T \) a syntactic object, is the object obtained by replacing each occurrence of \( v_i \) by \( t_i \), \( i = 1, \ldots, n \). Note that \( \vartheta = \nu \) iff \( \nu \notin \text{domain}(\vartheta) \). The product (or composition) of substitutions is defined so as to satisfy \( (T\vartheta)\sigma = T(\vartheta\sigma) \).

It is associative. \( \epsilon \) is the substitution with empty domain, i.e., the identity substitution. \( \vartheta \) is an idempotent substitution if \( \vartheta\vartheta = \vartheta \). This is equivalent to \( \text{domain}(\vartheta) \cap \text{range}(\vartheta) = \emptyset \).

A renaming is a substitution, \( \vartheta \), which permutes its domain. This is equivalent to the existence of \( \vartheta^{-1} \) satisfying that \( \vartheta^{-1}\vartheta^{-1} = \vartheta^{-1} = \epsilon \). A renaming of \( T \), called also a renamed variant of \( T \), is any \( T\vartheta \), where \( \vartheta \) is a renaming.

\( T \) and \( S \) are unifiable if, for some substitution \( \vartheta \), \( T\vartheta = S\vartheta \). Such a \( \vartheta \) is a unifier of \( T \) and \( S \). A most general unifier, \( \text{mgu} \) for short, is a unifier \( \vartheta \) such that any other unifier is of the form \( \vartheta\sigma \). Such a unifier exists if \( T \) and \( S \) are unifiable. Moreover, there exists an idempotent mgu. The domain and the range of idempotent mgus are included in \( \text{vars}(T) \cup \text{vars}(S) \). If \( \vartheta \) is an mgu of \( T \) and \( S \) then all other mgus are exactly all \( \vartheta\rho \) where \( \rho \) is a renaming. Mgus related thus are equivalent. In the case of idempotent mgus, \( \rho \) is a product of transpositions \( \{ u := v, v := u \} \) such that all \( u := v \) are bindings of \( \vartheta \) (cf. [LMM] for a most recent detailed account, for basic results cf. [Li]).

Henceforth we mean by "mgu" an idempotent mgu.

### Clauses and Resolutions

A clause is a disjunction \( A_1 \lor \ldots \lor A_m \lor B_1 \lor \ldots \lor B_n \), where the \( A_i \) and \( B_j \) are atoms, written also as: \( A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n \). \( A_1, \ldots, A_m \) is the head of the clause, \( B_1, \ldots, B_n \) — its body. The clause is negative if \( m = 0 \), definite — if \( m = 1 \), Horn — if \( m \leq 1 \).

Dually, a positive clause is one in which \( n = 0 \). We can also consider the dual notion of co-Horn clause \( (n \leq 1) \).

A unit clause is a clause of the form \( \leftarrow B \), \( A \leftarrow \). Occasionally we identify atoms with positive unit clause. A literal (or clause) is ground if it contains no individual variables.

The order of literals is significant in some concrete logic programming languages such as Prolog; in other, such as concurrent logic languages, it can be ignored. In general, we shall assume here multiset clauses, i.e., clauses in which we disregard the ordering. All our results are, however, valid also for ordered Horn clauses.

A set clause is a multiset clause in which every literal occurs at most once. \( \text{Set}(C) \) is the set clause obtained from \( C \) by deleting repeated occurrences (and deleting the order if \( C \) is ordered). Unions and subtractions are defined as usual for multisets.

If \( C_1 \) contains the literal \( L_i \), \( i = 1, 2 \), and \( L_1 \) is unifiable with \( L_2 \), then a (binary) resolvent of \( C_1 \) and \( C_2 \), obtained by resolving \( L_1 \) against \( L_2 \) (or upon \( L_1 \) and \( L_2 \)) is a clause \( (C_1\vartheta \setminus \{L_1\vartheta\}) \cup (C_2\vartheta \setminus \{L_2\vartheta\}) \) where \( \vartheta \) is an mgu of \( L_1 \) and \( L_2 \). The operation is called (binary) resolution.

Resolution thus defined is multiset resolution. It can produce multisets which are not sets even if \( C_1 \) and \( C_2 \) are sets. The set-resolution is defined by interpret-
ing the above expressions in terms of sets. It involves
the pruning of the result from repeated occurrences.
For ordered Horn clauses the resolvent is ordered by
replacing the resolved negative literal by the (ordered)
ing the above expressions in terms of sets. It involves
the pruning of the result from repeated occurrences.
For ordered Horn clauses the resolvent is ordered by
replacing the resolved negative literal by the (ordered)
body of the clause whose head is resolved. (cf. [LI])

A general resolution (also called resolution with fac-
toring, cf. [ChLe] for the case of set clauses) resolves a
multiset of positive literals against a multiset of nega-
tive ones. The resolvent is \((C_1\theta - D_1\theta) \cup (C_2\theta - D_2\theta)\),
where \(D_i\) is a non-empty submultiset of \(C_i\) and \(\theta\) is an
mgu of \(D_1\) with \(D_2\), i.e., with the multiset obtained
by negating each member of \(D_2\).

Resolution, unqualified, will refer to binary resolution
(for multiset or for ordered clauses).

Clauses are apart if they have no common variables.
Henceforth we assume (with no loss of generality) that
resolutions are applied only to clauses that are apart.

Provable Clauses

We use \(P Q, P', Q', P_1, \ldots\) to range over sets of clauses.

Definition 1 \(C\) is provable from \(P\), denoted \(P \vdash C\) if
\(C\) is in the minimal set containing renamings of
members of \(P\) and closed under resolutions.

We use "derivable" as synonymous with "provable".

The notion of a proof from \(P\) can be defined in any
of the standard ways. E.g., if proofs are trees, then a
proof from \(P\) is a binary tree with leaves labeled by
renamings of members of \(P\) and every internal node
labeled by a resolvent of the labels of its sons. It is
easily seen that for sets of Horn clauses the derivable
clauses are Horn, similarly for definite, co-Horn and
co-definite clauses.

3 Resolution Graphs

Recall that an undirected tree (known also as "free
tree", or "rootless tree") is a connected acyclic undi-
rected graph.

A resolution graph \(G\) is an undirected tree together
with:
(i) A labeling which labels each node \(x\) by a clause
\(C_x\).

(ii) An assignment which assigns each edge \(e = (x, y)\)
a literal-occurrence \(e(x)\) in \(C_x\) and a literal-occurrence
\(e(y)\) in \(C_y\). We say that \(e\) is attached to \(e(x)\) and \(e(y)\)
and that it links \(e(x)\) with \(e(y)\).

Moreover, the following conditions are required to hold:

(I) Clauses labeling different nodes are apart.

(II) If \(e\) and \(e'\) are different edges incident on the same
node \(x\), then \(e(x)\) and \(e'(x)\) are different occurrences
in \(C_x\).

(III) (Unifiability Condition) If \(e_1, \ldots, e_n\) are all the
edges of the graph, \(e_i = (z_i, z'_i)\), then
\((e_1(x_1), \ldots, e_n(x_n))\) is
unifiable with \((\bar{e}_1(z'_1), \ldots, \bar{e}_n(z'_n))\), where \(\bar{e}(x)\) is the
negated \(e(x)\).

See the introduction for an example of a resolution
graph.

The clause consisting of all unlinked literal-
occurrences is the remainder of \(G\), to be de-
noted \(\text{rem}(G)\). The mgu of \((e_1(x_1), \ldots, e_n(x_n))\) and
\((\bar{e}_1(z'_1), \ldots, \bar{e}_n(z'_n))\) is the substitution of \(G\), to be de-
noted \(\text{sub}(G)\). It is unique up to mgu equivalence. The
resolvent of \(G\), denoted \(\text{Res}(G)\), is \(\text{rem}(G)\text{sub}(G)\). It is
unique up to a renaming between equivalent mgu's.

To resolve a graph we carry out repeated resolutions;
for every edge \(e\) we form a resolvent of the clauses
labeling the end points by resolving upon the liter-
all-occurrences linked by \(e\). Here is the algorithm for re-
solving an edge \(e\), where \(e = (x, y)\):

1. Construct an mgu \(\theta\) of \(e(x)\) with \(e(y)\).

2. Collapse \(x\) and \(y\) into a single node \(z\) labeled by
\((C_x - \{e(x)\}) \cup (C_y - \{e(y)\})\). Make every other edge
\(e'\), which is incident either on \(x\) or on \(y\), incident on
\(z\) and attached to the same occurrence.

3. Change every literal \(c\) in \(C_x\) to \(c\theta\).

Remark 1: If \(x' \neq x\) and \(x' \neq y\) then \(c\theta = c\) for
every literal \(c\) of \(C_x\) (because all clauses are apart and
\(\theta\) binds only variables in \(C_x \cup C_y\)). Hence (3) can be
rephrased as: Change every literal \(c\) in the graph to
\(c\theta\).

Remark 2: Since \(\theta\)'s range contains only variables
occurring in \(C_x \cup C_y\), the clauses obtained after resolv-
ing \(e\) are apart.

Remark 3: So far we do not assume that the clauses
in question are Horn. The claims of this section, as well
as Lemma 7 of the next, hold for general clauses. How-
ever the completeness theorem is not valid for general
clauses if we allow only binary resolutions. It is valid
if we allow general resolutions and the proof, which
is analogous, uses a more general notion of resolution
graph which we shall not bring here.

We use \(\text{Res}(e, G)\) to denote any graph resulting from
\(G\) by resolving \(e\). (It is unique up to the choice of the
mgu \(\theta\).) The clauses of \(G\) are all \(C_x\) where \(x\) is a node
of \(G\). A resolution graph is over a set of clauses \(P\) if
its clauses are renamings of members of \(P\).
Theorem 1 If $G$ is a resolution graph and $e$ is an edge in $G$, then $\text{Res}(e, G)$ is a resolution graph. If moreover every clause of $G$ is provable from $P$, the same is true for $\text{Res}(e, G)$.

Proof The clauses in $\text{Res}(e, G)$ are apart, by Remark 2. The unifiability condition follows from Remark 1 and the fact that if $\alpha, \beta, \alpha', \beta'$ are syntactic objects such that $(\alpha, \beta)$ is unifiable with $(\alpha', \beta')$ and if $\theta$ is an mgu of $\alpha$ with $\alpha'$ then $\theta \beta$ is unifiable with $\beta \theta$. The second claim is trivial since the only new clause in $\text{Res}(G)$ is a resolvent of clauses of $G$.

Evidently, after resolving all edges we get a graph with a single node. We say that the clause labeling this node is a result of resolving the graph $G$.

Theorem 2 $\text{Pl-C}$ iff $C$ is a result of resolving some resolution graph over $P$.

Proof The "if" direction is obvious from Theorem 1. Evidently, renamed members of $P$ are results of resolving one-node graphs. It remains to show that the results of resolving graphs are closed under resolutions. Say, $C_1$ is obtained by resolving $G_1$, $i = 1, 2, \ldots$, where with $\text{no loser of generality}$ the $G_i$'s have no common nodes and no common variables in their clauses. Let $C$ be a resolvent of $C_1$ and $C_2$ where $C_1 \in C_2$ is resolved against $C_2 \in C_1$. Each occurrence in $C_1$ results from some unlinked occurrence in $G_1$ via some substitution. Say $e_i = C_1^{e_i} \in C_1$ is an occurrence in the clause labeling $x_i$ in $G_1$. Combine $G_1$ and $G_2$ to a single graph by connecting $x_1$ with $x_2$ by a new edge $e$ which links $C_1^{e_1}$ with $C_2^{e_2}$. For $i = 1, 2$ resolve the edges of $G_i$ so as to get $G_i$, then resolve $e$ so as to get $C$. □

The following easily proved theorem establishes the invariance of the final result under order permutations of edge-resolutions. It is used heavily throughout.

Theorem 3 (Order Independence) Let $G$ be a resolution graph and let $C$ be a result of resolving all the edges of $G$ in the order $e_1, \ldots, e_n$, where $e_i = (x_i, x'_i)$. Let $\theta_i$ be the mgu used in resolving of $e_i$ and let $\theta = \theta_1 \cdots \theta_n$. Then $C = \text{rem}(G) \theta$ and $\theta$ is a mgu of $(\tilde{e}_1(x'_1), \ldots, \tilde{e}_n(x'_n))$ with $(\tilde{e}_1(x'_1), \ldots, \tilde{e}_n(x'_n))$. Hence, modulo renamings between equivalent mgu's, $C$ is a resolvent of $G$.

Proof The resolving of an edge changes every undeleted literal-occurrence $c$ to $\phi c$, where $\phi$ is the mgu used in the resolving. Hence the result of resolving all edges changes each $c$ in $\text{rem}(G)$ to $c \phi_1 \cdots \phi_n$. The nondeleted occurrences are those of $\text{rem}(G)$ hence the result is $\text{rem}(G) \theta$.

To establish the second claim one shows by induction that $\theta_1 \cdots \theta_i$ is an mgu of $(e_1(x_1), \ldots, e_i(x_i))$ with $(\tilde{e}_1(x'_1), \ldots, \tilde{e}_i(x'_i))$. One uses the fact that if $\phi$ is an mgu of the syntactic object $a$ with $a'$ and $\mu$ is an mgu of $b \phi$ with $b' \phi$ then $\phi \mu$ is an mgu of $(a, b)$ with $(a'b')$.

Supplement Every mgu $\theta_1 \cdots \theta_n$ of $(\tilde{e}_1(x'_1), \ldots, \tilde{e}_n(x'_n))$ is obtainable as a product $\theta_1 \cdots \theta_n$, where $\theta_i$ is an mgu of $e_i(x_i) \theta_1 \cdots \theta_{i-1}$ with $\tilde{e}_i(x'_i) \theta_1 \cdots \theta_{i-1}$.

This can be derived (by a more delicate argument, not to be given here for reasons of length) from results in [LMM]. One can also show that, for a fixed ordering of the edges, this factorization of an mgu as a product of $\theta_i$'s is unique.

The Order Independence Theorem is not true for set resolutions.

Example Consider the following 3 clauses where $A$ is a 0-ary predicate and $f$ is a function symbol.

$C_1 = A \leftarrow R(z, y), P(z), P(y)$
$C_2 = R(fu, fu) \leftarrow$
$C_3 = P(fy) \leftarrow$

Resolving $C_1$ with $C_2$ and the outcome of this with $C_3$ yields $A \leftarrow$ . But if we resolve first $C_1$ with $C_3$, then resolving the outcome with $C_2$ gives $A \leftarrow P(fy)$.

Although the outcome of iterations of set resolutions is, in general, order dependent, resolution graphs can be quite useful for analyzing set resolutions and the possible effects of certain order permutations. They are used in deriving the completeness theorem for set resolutions.

3.1 Directed Resolution Graphs

It is sometimes useful to direct the edges of a resolution graph as follows:

For $e = (x, y)$, direct the edge from $x$ to $y$ if $e(y)$ is a positive literal, i.e., $e$ is attached to an atom in the head of $C_y$.

If the resolution graph is over Horn clauses then every $C_y$ contains at most one atom in the head, implying that every $y$ has at most one in-going directed edge. The connectedness of the graph implies that there is exactly one node without in-going edges. Hence the directed graph is a tree.
For co-Horn clauses we get a tree by directing in the opposite direction, i.e., from positive end points to negative ones.

Here is an application of the Order Independence Theorem and the directed version of resolution graphs:

An incremental proof of $C$ from $P$ is a sequence of clauses $C_1, \ldots, C_n$ such that $C_n = C$, $C_1$ is a renamed member of its sons. (Note that this is stronger than notion of clauses given). A which the literals resolved in each $C_i$ are in the body. Left sided proofs are defined dually.

**Theorem 4** If $P \vdash C$ then there is an incremental proof of $C$ from $P$. If, moreover, $P$ consists of Horn clauses there is a right-sided proof and if it consists of co-Horn clauses there is a left-sided proof.

**Proof** Let $C = Res(G)$, where $G$ is over $P$. Let $x_0, \ldots, x_n$ be an ordering of all nodes such that $x_0, \ldots, x_i$ is connected for all $i \leq n$ (since $G$ is connected there are such orderings). For each $0 < i \leq n$ there is a unique $j < i$ such that $(x_i, x_j)$ is an edge (the uniqueness is implied by the acyclicity of $G$). Let $e_i$ be this edge. Then $e_1, \ldots, e_n$ is an ordering of all edges. Resolving in this order we get after $i$ resolutions the value of the subgraph on the nodes $x_0, \ldots, x_i$. This value is then resolved with $C_{e_i}$. Thus we get an incremental proof.

In the case of Horn clauses let $G$ be a (directed) tree. Order the nodes as above so that, in addition, the sons of each node come after it. Resolving in this order gives a right-sided proof.

The argument for co-Horn clauses is dual. □

The claim concerning right-sided proofs is not true for general clauses. Theorem 4 does not hold for set resolutions.

**Example** Let $P$ consist of the following 4 clauses, where $A$, $A'$, $A''$ are 0-ary predicates.

\begin{align*}
C_1 &= A \leftarrow P(u, u, v), R(u), R(v) \\
C_2 &= R(v) \leftarrow A' \\
C_3 &= P(x, y, y) \leftarrow Q(z), Q(y) \\
C_4 &= Q(y) \leftarrow A''
\end{align*}

Resolving $C_1$ with $C_2$ we get $A \leftarrow P(u, u, v), R(u), A'$. Resolving $C_3$ with $C_4$ we get $P(x, y, y) \leftarrow Q(z), A''$. Resolving the two outcomes gives $A \leftarrow R(u), Q(u), A', A''$.

There is however no incremental set-resolution proof of this clause from $P$, because resolving $P(u, u, v)$ against $P(x, y, y)$ unifies $R(u)$ with $R(v)$ as well as $Q(z)$ with $Q(y)$. Any further resolution with $C_1'$ deletes the predicate $r$ altogether and any further resolution with $C_2'$ deletes $q$. Hence the resolving of $P(u, u, v)$ against $P(x, y, y)$ must be the last step if we want to have both $r$ and $q$ in the outcome.

**Ordered Horn Clauses**

All our results carry over to the case of ordered Horn clauses. For this purpose consider resolution graphs defined as before, except that the labels are ordered Horn clauses. Moreover, direct the graph as described above, so that it constitutes a tree. If $e = (x, y)$ links the head of $C_y$ with an occurrence in the body of $C_x$, the clause obtained by resolving $e$ is ordered as usual (cf. [Li])! Replace the literal occurrence, $e(z)$, in the body of $C_x$ by the ordered body of $C_y$ (and apply the mgu).

We now define a total ordering of all literal occurrences in this graph by interleaving clauses as follows:

If $e$ is an edge from $x$ to its son $y$, insert $C_y$ between $e(x)$ and its successor (or after $e(x)$ if $e(x)$ is the last occurrence in $C_y$).

The transitive closure of the $\leftarrow$-relation obtained by all such interleaving of pairs is a total ordering. Now define the remainder $rem(G)$ as the ordered clause consisting of all unlinked literal occurrences under the induced ordering.

It is easy to see that the resolving of edges preserves the total order just defined. Hence, in the end we get the remainder. Thus we have:

**Theorem 5** The Order Independence Theorem and all its consequences hold for ordered Horn clause resolutions.

4 The Completeness Theorem

Notation: $\forall C$ is the universal closure of $C$, obtained by quantifying universally over all vars($C$). For a set of clauses $P$, $\forall P$ is the set consisting of all $\forall C$, where $C \in P$.

We use $|=\neg$ for logical (semantic) implication:

$S \models H$, where $S$ is a set of wffs and $H$ is a wff, means that the satisfaction of all members of $S$ entails that of $H$, in all models which interpret the vocabulary of $S \cup \{ H \}$ and for every assignment of values to unquantified variables.

In the context of model theoretic semantics, $\forall C$ is the standard interpretation of the clause $C$ and $\forall P$ is the standard interpretation of the set of clauses $P$. We say that $P$ logically implies $C$ if $\forall P \models \forall C$

Evidently, not every $C$ logically implied by $P$ is provable from it. For example, if $P = \{ R(z) \leftarrow \}$
then only the clause itself, \( R(z) \leftarrow \) is provable from it, but among its logical consequences we have the following clauses, where \( S() \) is any predicate symbol and \( c \) is a constant:

\[
S(x) \leftarrow S(x) \quad R(x) \leftarrow \quad R(x) \leftarrow S(y)
\]

The first is a tautology, hence a logical consequence of every set of clauses, the second is an instance of \( S() \) and the third is obtained from \( P \) by weakening, i.e., by enlarging the disjunction. Essentially, our completeness theorem asserts that these are the only exceptions:

Every non-tautological consequence is obtainable from some provable clause by an instantiation followed by a weakening.

First some terminology:

A clause is tautological if it contains an atom and its negation.

A clause \( C \) subsumes \( C' \) if, for some substitution \( \theta \), every literal occurring in \( C' \) occurs in \( C \) (i.e., \( \text{set}(C \theta) \subseteq \text{set}(C') \)).

Evidently, if \( C' \) subsumes \( C \) then \( (\forall)C' \models (\forall)C \).

**Theorem 6 (Completeness and Soundness)** If \( P \) is a set of Horn clauses then \( (\forall)P \models (\forall)C \) iff either \( C \) is tautological or \( P \models C' \) for some clause \( C' \) which subsumes \( C \). This is also true for sets of co-Horn clauses.

The same is true for set-resolutions.

If \( P \) is a set of general clauses the claim is true provided that we use general resolutions instead of binary resolutions. This is also true for general resolutions of set clauses.

The "if" direction is the easily established soundness of the resolution rule. The major lemma used in the completeness proof is the following Deduction Lemma.

**Notation**: The empty clause is denoted by \( \bot \).

For a set of clauses \( P \) and for literals \( L_1, \ldots, L_n \) we use \( P, L_1, \ldots, L_n \) for the set obtained by adding to \( P \) the indicated literals, where we regard every atom \( A \) as \( A \leftarrow \) and every negated atom \( \overline{A} \) as \( \overline{A} \leftarrow \).

**Lemma 7 (Deduction)** Let \( L_1, L_2, \ldots, L_n \) be ground literals. Then \( P, L_1, \ldots, L_n \models \bot \) iff either \( L_i = \overline{L}_j \) for some \( i, j \), or there exists a \( C \) subsuming \( L_1 \vee \ldots \vee L_n \) such that \( P \vdash C \).

This is the Resolution Logic version of the Deduction Theorem for first order logic. There is also a more general form in which the \( L_i \) are not assumed ground, but we do not need it here.

**Proof (Outline)** The "if" direction is straightforward. For the other direction, consider a resolution graph over \( P \cup \{L_1, \ldots, L_n\} \) whose resolvent is \( \bot \). If the set of added literals does not contain a literal and its negation there must be clauses in the graph which are (renamed) members of \( P \). The nodes labeled with (renamed) members of \( P \) constitute a connected subgraph. Resolving first all edges in this subgraph gives the desired \( C \).

**Remark**: The Deduction Lemma is also true for set resolutions, i.e., we can replace everywhere \( \models \) by \( \models \). As in the multiset case the proof is based on postponing the resolutions of the literals from \( \{L_1, \ldots, L_n\} \) to the end. Since the Order Independence theorem is no longer true, one has to analyse the possible effects of such a permutation, making for a more difficult proof.

The Lemma is also true for general resolutions, as well as for general resolutions of set-clauses (i.e., we can subscript everything either by \( g \) or by \( gs \)); here the more general form of resolution hypergraphs is used. Once the lemma is established, completeness is proved much in the same way in each of these cases.

**Completeness Proof (Outline)**

Given \( P \) and \( C \) as above, assume that \( C \) is non-tautological and that no clause derivable from \( P \) subsumes it. We construct a model satisfying \( (\forall)P \) but not \( (\forall)C \). We first reduce the case to that of a ground \( C \) by replacing the variables in \( C \) by distinct new constants not occurring in \( P \) or in \( C \). It is easily seen that also this \( C \) is not subsumed by any clause derivable from \( P \). So assume \( C = L_1 \vee \ldots \vee L_n \) where the \( L_i \) are ground. Define a set of clauses to be consistent if \( \bot \) is not provable from it. \( C \) is non-tautological and is not subsumed by any clause provable from \( P \), hence, by the Deduction Lemma, \( P \cup \{L_1, \ldots, L_n\} \) is consistent. Let \( P_0 \) be this set. (If \( C = \bot \), \( P = P_0 \).)

Now, if \( Q \) is any consistent set of Horn clauses and \( A \) is a ground atom then either \( Q \cup \{A\} \) or \( Q \cup \{\overline{A}\} \) is consistent. Otherwise, by the Deduction Lemma we can prove from \( Q \) a clause \( C_1 \) subsuming \( A \leftarrow \) and we can also prove a clause \( C_2 \) subsuming \( \overline{A} \leftarrow \overline{A} \). Since \( Q \) consists of Horn clauses, \( C_1 \) must be a positive unit clause. We can use \( C_1 \) to knock off, by binary resolutions, all members in the body of \( C_2 \), getting eventually \( \bot \), which contradicts the consistency of \( Q \). A dual argument works for sets of co-Horn clauses. (For general clauses we can get \( \bot \) by a general resolution which resolves the head of \( C_1 \) against the body of \( C_2 \). It is at this point that we must use non-binary resolutions in the general case).

Starting with \( P_0 \) we can now add repeatedly ground literals, a literal at a time, so as to get in the end a consistent set containing for each ground atom \( A \) either \( A \) or its negation. Such a "maximal" consistent set determines a model satisfying \( (\forall)P \) but not \( (\forall)C \). (More details in the full paper.)

**Theorem 6** implies that an atom is a logical consequence of \( P \) iff it is an instance of a provable atom. Using the equivalence between provability of atoms from programs and derivability via SLD refutations.
(the forthcoming Theorem 8) we get the known completeness results of [CI] and [Hi] (cf. also [ApEm], and [Li]). The independence of the derivability on the selection rule (what is known as "strong completeness") is an immediate corollary of the Order Independence Theorem. Moreover, Robinson's completeness result [Ro] is an immediate corollary of the general-resolution version of the theorem for the case where $C = \text{false}$. Thus, our Completeness Theorem generalizes all the known results from single atoms to clauses.

5 Answer Substitution and Derivable Atoms

Henceforth, clause, unqualified means a Horn clause.

A logic program, or program, for short, is a set of definite clauses. Given a program $P$ an SLD refutation of $\neg A$, where $A$ is an atom, consists of resolving $\neg A$ with some renamed member of $P$, followed by a sequence of resolutions with renamed members of $P$, whose end result is false. If $\theta$ is the product of the mgu's used in the refutation, then its restriction to $\text{vars}(A)$ is the answer substitution of the refutation. $\neg A$ is referred to as "atomic goal" in the accepted terminology. Actually, the refutation does not establish the goal but an instance of its negation, namely $A \sigma$, where $\sigma$ is the answer substitution. The significance of the method lies in the fact that $(\forall)A \sigma$ is a logical consequence of $(\forall)P$ (this is the soundness of SLD resolutions).

Here we are interested in direct proofs, rather than proofs by refutation and we show how to get such direct proofs and how they determine the answer substitutions. We therefore describe the situation positively and say that $A$ is SLD-derivable from $P$ with answer substitution $\sigma$.

If the refutation starts with $\neg A_1, \ldots, A_n$ (a so called "conjunctive goal"), instead of $\neg A$, then the answer substitution, $\sigma$, is the restriction of $\neg A$ to $\text{vars}(A_1, \ldots, A_n)$. In that case we have actually derived from the program (the universal closure of) $(A_1 \land \cdots \land A_n) \sigma$. Again, we describe it positively by saying that $A_1 \land \cdots \land A_n$ is SLD-derivable with answer substitution $\sigma$.

Two programs may have the same atomic (positive unit clause) logical consequences, and even be logically equivalent, without giving the same answer substitutions. For example, the programs $\{P(x) \leftarrow \}$ and $\{P(z) \leftarrow, P(c) \leftarrow\}$, where $c$ is a constant, are logically equivalent; on the goal $\neg P(x)$ the first can give only $x = c$ (which leaves $x$ intact) as answer substitution but the second can give also the more particular case $x := c$. Evidently, the possible answer substitutions reflect operational details not captured by the program's logical consequences. It turns out that the information conveyed by the pairs $(\text{goal}, \text{answer-substitution})$ is exactly the information given by the set of all derivable atoms: $\{A : Pt \leftarrow A\}$. Thus the distinction between the logically oriented model-theoretic semantics and the more operational proof-theoretic semantics is precisely captured by the distinction between $\models$ and $\vdash$.

Theorem 8 $A$ is SLD-derivable from $P$ with answer substitution $\sigma$ iff $Pt \leftarrow A'$, for some $A'$ unifiable with $A$ via an mgu whose restriction to $\text{vars}(A)$ is $\sigma$.

This generalizes to conjunctions: $A_1 \land \cdots \land A_n$ is SLD-derivable with answer substitution $\sigma$ iff $Pt \leftarrow A'$, for some $A'_1, \ldots, A'_n$ such that $(A'_1, \ldots, A'_n)$ is unifiable with $A_1, \ldots, A_n$ via an mgu whose restriction to $\text{vars}(A_1, \ldots, A_n)$ is $\sigma$.

Proof (Outline) An SLD refutation determines in the obvious way a right-sided proof of $false$ which starts with $\neg A$ (or with $\neg A_1, \ldots, A_n$ for conjunctive goals). We get a corresponding resolution graph which is a tree (cf. 3.1). Its root is labeled by the negative goal-clause. Let us resolve first all edges which do not issue from the root. This yields the derivable clause $A'$ unifiable with $A$. (In the conjunctive case we get the derivable clauses $A'_i$, $i = 1, \ldots, n$). If $\theta'$ is the mgu of the derivation of $A'$ then the mgu of the refutation is $\theta' \rho$, where $\rho$ is an mgu of $A$ and $A'$. The restrictions of $\theta' \rho$ and of $\rho$ to $\text{vars}(A)$ are the same, because the resolution tree clauses are apart, implying that $\text{vars}(A)$ is disjoint from the domain of $\rho$. That every substitution answer is thus obtainable follows from The Supplement to the Order Independence Theorem. The conjunctive case is similarly established.

The other direction is symmetric: If $Pt \leftarrow A'$, then some resolution tree over $P$ resolves to $A'$ and we can get from it a corresponding SLD-refutation. (In the conjunctive case the SLD-refutation is obtained, using the resolution trees for $A'_i$, $i = 1, \ldots, n'$.)

Corollary 1 For any two programs $P, Q$ the following are equivalent:

(i) $P$ and $Q$ have the same derivable atoms.

(ii) For every atomic goal, $P$ and $Q$ give the same answer substitutions.

(iii) For every conjunctive goal, $P$ and $Q$ give the same answer substitutions.\(^2\)

Theorem 8 provides a direct definition of "answer substitution", via derivability of atoms, as an alternative to the usual roundabout way of defining it (cf. [Li]).

\(^2\)To show directly the equivalence of (ii) and (iii) is not difficult, but somewhat messy. It has been probably observed before, though we could not find a reference.
One natural choice of observables are the answer substitution pairs, i.e., the pairs \((\text{goal, answer-substitution})\) that go with a program. If this is our choice and we do not require our semantics to be invariant under certain types of compositions (i.e., we take an empty set of program-compositions) then the set of provable atoms can serve as a fully abstract semantics. This semantics has been obtained in [FLMP] in a different way, not based on a proof system.\(^3\) Deriving the semantics from a proof system puts it in a wider perspective and leads to generalizations which are compositional for various types of program compositions. Provable atoms are sufficient as long as we do not have compositionality requirements, but even the simplest type of compositions (such as adding atoms to a program) forces us to consider provable clauses and it is here that the full power of the proof theory is required. (An analogous situation arises in the logical-consequence based semantics, cf. [GaSh].)

6 Compositions

We fix our observables to be the pairs \((\text{goal, answer-substitution})\) associated with a program. Equivalently, we can take them to be the program's derivable atoms. We shall now consider various program compositions.

There are two types of compositions: unstructured and modular. In both types the composition of two (or more) programs is their union, but in the modular case the programs have additional structure which makes them into what we call logic modules. Logic modules and their compositions have are defined in [GaSh].

A logic module is a logic program with a classification of its predicates into three exclusive classes: \textit{Imported}, \textit{Exported} and \textit{Internal}. Imported predicates cannot occur in the head of any clause. The idea is that the imported predicates are not computed by the module, they are supplied (as sets of atoms \(P(t_1,\ldots,t_n)\)) by the environment (the user, or other modules). Exported predicates are computed by the module and may be supplied to the environment. Internal predicates are local to the module and hidden from the environment. They can be renamed without affecting the module's functioning.

The program of the composition of logic modules is the union of their programs; the composed modules may share predicates, subject to the following restrictions:

(i) An internal predicate of a module cannot occur in any other module.

(ii) An exported predicate of a module can occur in another module only if it is imported there.

When an exported predicate is shared by another module this means that the second module draws this import from the first. When the same predicate is imported in several modules it means that all share the same source of this import.

A predicate is classified as exported in the composition iff it is exported in one of the composing modules. It is imported in the composition if it is imported in some module and not exported in any other. All the rest are internal.

The module algebra has also a \textit{hide} operation, by which an exported predicate becomes internal. However our results do not concern it.

Modular compositions have obvious meaning in terms of software engineering. Not so unstructured compositions (simple unions) whose interest is mainly mathematical. However the results concerning the unstructured case carry over to the modular case by relativizing to \(I/E\)-clauses.

For a given logic module, \(C\) is an \(I/E\)-clause (Import/Export clause) if all its body predicates are imported and its head predicate is exported. Intuitively, it is to be expected that a module's interaction with any environment is determined only by its derivable \(I/E\)-clauses (the environment cares only about the relation from import to export, not how this relation is computed). This is indeed the case, though it takes some proving.

By "derivable from a given module" we mean derivable from the module's program.

An \textit{export atom} of a module is an atom whose predicate is exported. \textit{Import atom} and \textit{internal atom} are similarly defined.

The \textit{observables} of a module is the set of all its derivable export atoms or, equivalently, all the answer-substitution pairs for export atoms. (Note that no import atom is derivable from a module.)

In what follows we state the results for the unstructured case and then the corresponding modular version. The proofs employ the resolution-graph technique. We include the proof of the unstructured version of Theorem 11 as an illustrative example, the rest will be given in the full paper.

Consider first unions of a program with sets of ground atoms. \(P\) and \(Q\) are indistinguishable in the resulting semantics if for all sets \(V\) of ground atoms (i.e., ground positive unit clauses) \(P \cup V\) and \(Q \cup V\) have the same derivable atoms.

[In the case of a logic module \(M\), adding an atom \(A\) means composing \(M\) with the module \(M' = \{A--\}\) in which the predicate, say \(R\), of \(A\) is exported. (If

\(^3\)It is the least fixpoint of a certain operator over a Herbrand base that includes non-ground atoms.
R is internal in M, rename the internal predicates of M.) Note that if R is not imported in M then the composition will have only A as an additional derived atom.

We recall that in the logical-consequence based semantics, where the observables are the logically implied atoms, indistinguishability under ground atoms addition implies indistinguishability under addition of any clauses (cf. [GaSh]). Here this is no longer true: By the forthcoming Theorem 9 the following programs are indistinguishable by ground atom addition:

\[
P_1: \quad S(v) \leftarrow R(v, u)
\]

\[
P_2: \quad S(v) \leftarrow R(v, u), \quad S(fu) \leftarrow R(fu, u)
\]

But they can be distinguished if we add to them \(R(z, y)\): we can derive \(S(fy)\) from the first but not from the second.

**Definition 2** A clause \(C\) is b-subsumes \(C'\) if \(C\) subsumes \(C'\) via a substitution which binds only variables occurring in the body of \(C\).

**Theorem 9** Two programs \(P\) and \(P'\) are indistinguishable by ground atom additions iff every clause provable from \(P\) is b-subsumed by some clause provable from \(P'\) and vice versa.

For logic modules the same is true if we replace every “clause” by “I/E-clause” (referring to I/E-clauses in the respective module).

**Terminology:** \((\forall b)C\) is the result of quantifying universally all the variables of the body of \(C\).

Say that \(C\) properly b-subsumes \(C'\) if \(C\) b-subsumes \(C'\) but not vice versa.

If \(Q\) is any set of clauses, say that \(C \subseteq Q\) is b-minimal in \(Q\) if it is not properly b-subsumed by any clause in \(Q\) with a smaller or equal number of literals. (It is not difficult to see that every member of \(Q\) is b-subsumed by some b-minimal member.)

A b-instance of \(C\) is any instance \(C \theta\) where \(\theta\) binds only variables in the body of \(C\).

**Theorem 10** The following are equivalent:

1. \(P\) and \(P'\) are indistinguishable by ground atoms addition.
2. \((\forall b)C : P\), and \((\forall b)C : P'\) are logically equivalent.
3. \(\{\text{set}(C) \subseteq C\} : \{\text{set}(C) \subseteq C\} : \{\text{set}(C) \subseteq C\} : \{\text{set}(C) \subseteq C\}\) are indistinguishable.

The same is true for logic modules provided that in each case \(C\) ranges over the respective I/E-clauses.

We can take as invariant the set of all b-minimal members of \(\{\text{set}(C) \subseteq C\}\) is a b-instance of a clause provable from \(P\) \}. Again, for modules, restrict \(C\) to I/E-clauses.

It can be shown that this invariant is always in the program’s vocabulary. (Note that two indistinguishable programs need not be in the same vocabulary, e.g., \(\{S(v) \leftarrow R(v, u)\}\) and \(\{S(v) \leftarrow R(v, u), S(u) \leftarrow R(c)\}\), where \(c\) is an individual constant.) Thus the invariant is in the intersection of all vocabularies of programs indistinguishable by ground atoms addition.

Now consider addition of arbitrary atoms. By Theorem 11, programs \(P, Q\) indistinguishable by atom additions are also indistinguishable by clause addition, that is to say, \(P \cup Q\) and \(Q \cup W\) have the same derivable atoms, for all programs \(W\). It is not difficult to show that this indistinguishability is an equivalence relation, say \(\equiv\), which is also a congruence for unions: if \(P_i \equiv Q_i, i = 1, 2, P_1 \cup P_2 \equiv Q_1 \cup Q_2\).

**Theorem 11** Two programs (logic modules) which are indistinguishable by atom additions (compositions with exported atoms) are indistinguishable by unions (module compositions).

**Proof.** (for the case of unions) Let \(P\) and \(Q\) be indistinguishable by atom additions. Show that, for any program \(W\), every atom derivable from \(P \cup W\) is also derivable from \(Q \cup W\).

Consider a resolution tree \(G\) over \(P \cup W\) which resolves to an atom \(A\). We prove the claim by induction on the number \(n\) of nodes labeled by renamed members of \(W\). For \(n = 0\) the claim is trivial. Assume that \(n > 0\) and the claim holds for all smaller values.

Let \(C_x\) (the clause labeling \(x\)) be a renaming of a member of \(W\). Let \(G_x\) be the subtree whose root is \(x\). Then \(G_x\) resolves to an instance, say \(B\) of the head of \(C_x\). Let \(G'\) be the tree obtained by deleting all descendants of \(x\) and relabeling \(x\) by \(B\). By Order Independence \(G'\) resolves to \(A\).

Now \(G'\) is over \(P \cup \{B\} \cup W\) and has \(< n\) nodes labeled by renamed members of \(W\). Also \(P \cup \{B\}\) and \(Q \cup \{B\}\) are indistinguishable by atom additions (since \(B\) is an atom). By the induction hypothesis \(A\) is derivable from \(Q \cup \{B\} \cup W\).

We shall be done if we show \(Q \cup W = B\). Let \(G_1, \ldots, G_m\) be the the subtrees of \(G_x\) which hang from the sons of \(x\). By order independence we can first resolve these trees, getting atoms, say \(B_1, \ldots, B_m\) and then resolve each \(B_i\) with the atom in the body of \(C_x\).

Since every \(G_i\) has \(< n\) nodes labeled by renamed members of \(W\), we can derive each \(B_i\) from \(Q \cup W\).

4 If \(P_i \equiv Q_i\), then also \(P_i \cup P_i' \equiv Q_i \cup P_i'\), because \((P_1 \cup P_i') \cup W = P_1 \cup (P_i' \cup W)\) and similarly for \(Q_i\). We get therefore: \(P_1 \cup P_2 \equiv Q_1 \cup Q_2\).
and then derive $B$. □.

To get a fully abstract invariant for unstructured and modular program compositions we use the following notion:

**Definition 3** Let $C$ be a clause. An associated implication of $C$ is a pair $(A', D)$, such that $A'$ is an atom, $D$ is a set of atoms and \( \{C\} \cup D \vdash A' \).

The associated implications of a set of clauses are the associated implications of its members.

By the vocabulary of a program we mean the list of predicates and function symbols that occur in the program.

**Theorem 12** If $P$ and $P'$ are programs whose vocabularies are included in $V$, then $P$ and $P'$ are indistinguishable by unions iff \( \{C : P \vdash C\} \) and \( \{C : P' \vdash C\} \) have the same associated implications in the vocabulary $V$.

The same holds for logic modules provided that in each of the above mentioned sets $C$ ranges over the respective I/E-clauses.

This invariant may take us out of the program's vocabulary. It presupposes an inclusive list of function symbols in which all programs partake. We conjecture that the choice of associated implications can be restricted so as to remain within each program's vocabulary (and thus in the intersection of the vocabularies of equivalent programs).

**Acknowledgements**

We thank John Gallagher for very helpful comments on an earlier draft of the paper.

**Added in Press:**

The following two unrelated points have been brought to our attention:

1. A result similar to our completeness theorem (Theorem 6) has been stated in Robert Kowalski’s Ph.D. Thesis (Studies in the Completeness and Efficiency of Theorem Proving by Resolution, Edinburgh University, 1970).

2. The idea of linking literal occurrences by edges underlies Kowalski's connection graphs (A Proof Procedure Using Connection Graphs, J.ACM Vol. 22 No. 4, 1975, pp.572-595). Thus resolution graphs are related to this notion. Nonetheless they differ in other basic aspects. Connection graphs are differently constructed and applied, and do not have the properties of resolution graphs used in this paper, in particular the order independence property. In a later unpublished work (A Theorem Proving Approach to Database Integrity, F.Sadri and R. Kowalski) an evolved notion of connection graphs is used which is considerably closer to resolution graphs.

**References**


Hill, R., LUSH resolution and its completeness, DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.


