Chapter 1

Logic Programming: A Case Study

A semantics of logic programming can be derived from the Skolem-Herbrand-Gödel Theorem of the first-order predicate calculus.

We trace the developments which led to logic programming from automations of theorem proving which used this fundamental result, especially those based on the resolution principle. We also discuss some criteria by which logic programming can be considered to be a programming language, and some of its limitations.

This case study motivates our approach to higher-order logic programming: a higher-order Skolem-Herbrand-Gödel Theorem would give a basis for a more expressive logic programming language.

1.1 From Theorem Provers to Logic Programming

A main task of automated theorem provers is to find whether or not a formula A_{n+1} is a logical consequence of a conjunction of formulas $A_1 \wedge \cdots \wedge A_n$. For first-order classical logic [28], this is a recursively undecidable problem in general [25, 26], however there are semi-decision procedures for testing the validity of a formula which will halt if the formula being tested is valid, but may not halt if the formula is not valid.

Automated theorem provers have been used to answer questions such as "Does there exist a finite semigroup which simultaneously admits a non-trivial antiautomorphism without admitting a non-trivial involution?" [202], to prove Cantor's Theorem [9] and others in set theory, and to verify the correctness of a microprocessor [33].

1.1.1 The Skolem-Herbrand-Gödel Theorem

A fundamental result used in some of these procedures is the Skolem-Herbrand-Gödel Theorem [125][1].

We use the semantics of classical first-order logic [7] in the following discussion. The Skolem-Herbrand-Gödel Theorem concerns the validity of first-order formulas:

Theorem 1.1 *(Skolem-Herbrand-Gödel.) A formula A is valid if and only if a compound instance of a Skolem normal form of $\neg A$ is unsatisfiable.*

[1] See the papers by Skolem, Herbrand, and Gödel [84]. The Theorem is also called the Expansion Theorem, or incorrectly referred to as Herbrand's Theorem [88] which is a result in proof theory [49, 71].

The Skolem-Herbrand-Gödel Theorem depends on four other theorems. By the deduction theorem for classical first-order predicate calculus, showing

$$A_1 \wedge \cdots \wedge A_n \models A_{n+1}$$

is equivalent to showing

$$\models A_1 \wedge \cdots \wedge A_n \supset A_{n+1}.$$

We need only consider the validity of a formula, rather than showing that it is a logical consequence.

There is also no loss of generality if we only consider the unsatisfiability of a formula in Skolem normal form. This is a consequence of a theorem due to Skolem [63, 179] which states that for every first-order formula F there is a first-order formula F' in Skolem normal form such that

- F' is satisfiable if and only if F is satisfiable.
- F' can be effectively computed from F.

It follows that a formula is valid if and only if the Skolem normal form of its negation is unsatisfiable. The algorithm for transforming a formula to one in Skolem normal form is discussed in Section 1.1.2.

The significance of Skolem normal form depends on the following result [63, 121].

Theorem 1.2 *A formula in Skolem normal form has a model if and only if it has a Herbrand model.*

Herbrand models are also called free models [121] and term models. It follows that an arbitrary first-order formula is valid if and only if the Skolem normal form of its negation has no Herbrand model. The class of Herbrand models is smaller than the class of arbitrary first-order models because they are defined using the constant symbols which only occur in the Skolem normal form of a formula, and possibly one additional individual constant symbol.

This reduction makes automating testing validity possible since the absence of a Herbrand model can be tested by finding whether the Herbrand expansion of a formula in Skolem normal form is unsatisfiable. Herbrand expansions are discussed in Section 1.1.3.

The fourth necessary result to do this is the compactness theorem of propositional logic which ensures that the Herbrand expansion of a Skolem normal form formula is unsatisfiable if and only if one of its compound instances is unsatisfiable. Such compound instances can be recursively enumerated.

Major forms of automated theorem proving in first-order logic such as resolution [124], matings [6], and the connection graph method [8, 17] are based on the Skolem-Herbrand-Gödel Theorem and refinements of it which do not require the negation of a formula to be converted to a normal form. We now present resolution theorem proving as an example, before considering how it has been used in logic programming.

1.1.2 Skolem Normal Form

A formula in Skolem normal form has the form

$$\forall x_1 \cdots \forall x_n \, (C_1 \wedge \cdots \wedge C_m)$$

where the C_i are finite disjunctions of literals for $1 \leq i \leq m$, and $x_1, \ldots, x_n$ are the free variables in $C_1, \ldots, C_m$.

We shall convert the following formula F to its Skolem normal form F'.

$$\neg\exists x \forall y \forall z \neg[(\neg q(x,y) \equiv \exists t[\neg\forall y(q(x,y) \supset \neg p(f(v)))]) \wedge \forall x \neg q(v,x)]$$

Firstly, we take the existential closure of F, eliminate redundant quantifiers, and rename quantified variables so that they are distinct. We obtain:

$$\exists v \neg\exists x \forall y \neg[(\neg q(x,y) \equiv [\neg\forall z(q(x,z) \supset \neg p(f(v)))]) \wedge \forall w \neg q(v,w)]$$

We then eliminate logical constants other than $\neg, \wedge, \vee, \exists$ and $\forall$, and move negations all the way inwards to obtain:

$$\begin{array}{c}\exists v \forall x \exists y[[q(x,y) \vee [\exists z(q(x,z) \wedge p(f(v)))]] \wedge \\ [\neg q(x,y) \vee [\forall z(\neg q(x,z) \vee \neg p(f(v)))]] \wedge \forall w \neg q(v,w)]\end{array}$$

We then move quantifiers to the right, and eliminate $\exists$ by the introduction of Skolem constants. This gives:

$$\begin{array}{c}\forall x[[q(x,g(x)) \vee [q(x,h(x)) \wedge p(f(a))]] \wedge \\ [\neg q(x,g(x)) \vee [\forall z(\neg q(x,z) \vee \neg p(f(a)))]] \wedge \forall w \neg q(a,w)]\end{array}$$

Finally, we move $\forall$ to the left, distribute $\wedge$ over $\vee$, and simplify by deleting any tautologous conjuncts and repeated literals in conjuncts to obtain F':

$$\begin{array}{rcl} F' & = & \forall x \forall z \forall w[[q(x,g(x)) \vee q(x,h(x))] \wedge [q(x,g(x)) \vee p(f(a))] \wedge \\ & & [\neg q(x,g(x)) \vee \neg q(x,z) \vee \neg p(f(a))] \wedge [\neg q(a,w)]] \end{array}$$

The main steps in forming F' from F are the elimination of logical constants other than $\wedge, \vee$ and $\neg$, the elimination of existential quantifiers by the introduction of Skolem functions, and conversion of a subformula to conjunctive normal form. The process of removing quantifiers by introducing terms containing new function symbols is sometimes called Skolemization. Skolem functions are also called Herbrand-Skolem functions because of the extensive use made of them by Herbrand [88].

1.1.3 Compound Instances

The set of function symbols $\{a, f, g, h\}$ each of which occurs in the formula F' is a signature of function symbols. If such a signature does not contain a constant symbol such as a, one can be added. From this signature, a set of ground terms can be recursively enumerated. It includes the terms

$$a, f(a), g(a), h(a), f(f(a)), f(g(a)), f(h(a)), \ldots$$

This set of ground terms is called the Herbrand Universe of F'.

A Herbrand expansion of a formula in Skolem normal form is an infinite conjunction of Herbrand instances, each of which is formed by replacing every occurrence of a universally quantified variable by a term in the Herbrand Universe, and by deleting its universal quantifier.

Ground positive literals in a Herbrand expansion can be regarded as atomic propositions, and negative literals as negated atomic propositions. A finite conjunction of Herbrand instances of a formula is called a compound instance of the formula. For example,

$$[[q(a,g(a)) \vee q(a,h(a))] \wedge [q(a,g(a)) \vee p(f(a))] \wedge$$
$$[\neg q(a,g(a)) \vee \neg q(a,h(f(a))) \vee \neg p(f(a))] \wedge [\neg q(a,h(a))]] \wedge$$
$$[[q(f(a),g(f(a))) \vee q(f(a),h(f(a)))] \wedge [q(f(a),g(f(a))) \vee p(f(a))] \wedge$$
$$[\neg q(f(a),g(f(a))) \vee \neg q(f(a),g(h(a))) \vee \neg p(f(a))] \wedge [\neg q(a,h(h(a)))]]$$

is a compound instance of F'.

By the Compactness Theorem for propositional logic, the Herbrand expansion of a formula is unsatisfiable if and only if there is a compound instance of the formula which is unsatisfiable.

It is decidable whether a compound instance is unsatisfiable; a truth table can be constructed. Moreover the set of compound instances is recursively enumerable.

1.1.4 Testing Validity

By the deduction theorem a formula A_{n+1} is a logical consequence of hypotheses $A_1 \wedge \cdots \wedge A_n$ if and only if $A_1 \wedge \cdots \wedge A_n \wedge \neg A_{n+1}$ is unsatisfiable. This suggests the following procedure for testing whether A_{n+1} is a logical consequence of the hypotheses.

1. Form a Skolem normal form F' of $A_1 \wedge \cdots \wedge A_n \wedge \neg A_{n+1}$.
2. Repeatedly generate the next compound instance of F' and test if it is unsatisfiable. If so, halt with the answer 'valid'.

Since validity is recursively undecidable, this is a partial decision procedure which may never terminate if the Herbrand expansion of F' is satisfiable. If it halts, the Herbrand expansion of F' is unsatisfiable and A_{n+1} is a logical consequence of $A_1 \wedge \cdots \wedge A_n$.

Apart from its applications to automated theorem proving, as an indication of its generality we now state three major results of classical first-order predicate calculus which are direct consequences of the Skolem-Herbrand-Gödel Theorem [121].

Theorem 1.3 *The set of valid formulas and the set of unsatisfiable formulas of first-order predicate calculus are recursively enumerable.*

Proof: It suffices just to consider the case of valid formulas because a formula is valid if and only if its negation is unsatisfiable. The preceding procedure is suitable for testing validity. □

The next theorem is a special form of the Löwenheim-Skolem Theorem [178].

Theorem 1.4 *Every countably infinite set of formulas which has a model, has a model whose domain of individuals is countable.*

Proof: We can assume without loss of generality that the Skolem normal forms of the negations of the formulas have no constant symbols in common. There are countably many such constant symbols. The Herbrand expansion of each of the formulas must be satisfiable and any truth assignment which verifies all of the expansions immediately gives a countable Herbrand model. □

The third major consequence is the compactness theorem of first-order predicate calculus.

Theorem 1.5 *A countably infinite set of formulas has a model if and only if every finite subset of it has a model.*

In addition, apart from these direct consequences, the Theorem and the corrected version of Herbrand's Theorem [49, 88] imply Gödel's Completeness Theorem of first--order predicate calculus [72].

1.1.5 Unsatisfiability Procedures

For automated theorem proving to be practicable, it must be efficient. Systematically generating compound instances and testing their unsatisfiability is an obvious inefficiency in the previous procedure. Unsatisfiability procedures have been developed to reduce this inefficiency.

Formulas in Skolem normal form can be abbreviated to a form called clause form, which is simpler. This is done by replacing a disjunction of literals by a set containing the literals in the disjunction, by replacing a conjunction of such sets by a set containing the sets in the conjunction, and by removing all universal quantifiers. The set is called a formula, and its elements are called clauses. Renaming variables depends on the identity

$$\forall x_1 \cdots \forall x_n (C_1 \wedge C_2) \equiv (\forall x_1 \cdots \forall x_n C_1) \wedge (\forall x_1 \cdots \forall x_n C_2).$$

A clause is an empty clause, written □, if it is the empty set. A clause form formula is unsatisfiable if it includes the empty clause. A clause form formula which is the empty set is always satisfiable.

For example, the previous formula F' has the clause form:

$$\{\{q(x,g(x)), q(x,h(x))\}, \{q(x,g(x)), p(f(a))\},$$
$$\{\neg q(x,g(x)), \neg q(x,z), \neg p(f(a))\}, \{\neg q(a,w)\}\}$$

Davis and Putnam's method [42] for testing the unsatisfiability of a compound instance works directly with ground clauses; the abbreviated counterpart of compound instances. It is a direct implementation of the procedure above based on the Skolem-Herbrand-Gödel Theorem, and it is very inefficient. Attempts by Prawitz [160] and Davis [43] to improve its efficiency led to the resolution method of Robinson [166].

In its ground form, resolution extends one of the three rules of Davis and Putnam's method; the one-literal rule. The rule states that a formula is unsatisfiable if it includes

two singleton clauses $\{P\}$ and $\{\neg P\}$. Ground resolution can be seen as a generalization of this rule and a form of Gentzen's cut rule [70] or generalized *modus ponens*. It uses a single rule of inference which is called the ground resolution principle:

> *Choose any two clauses C_1 and C_2 in a set of ground clauses one of which contains a literal P and the other which contains a literal $\neg P$. Include the resolvent clause $C_1 \cup C_2 - \{P, \neg P\}$ in the set of clauses.*

The clause $C_1 \cup C_2 - \{P, \neg P\}$ is called the ground resolvent of the chosen clauses. The initial set of ground clauses is unsatisfiable if and only if after a finite number of applications of the ground resolution principle, the empty clause is a resolvent. The process of forming ground resolvents always terminates with a set which either contains the empty clause, or which contains all possible resolvents. Although ground resolution only uses one rule of inference, it is still impractical for theorem proving based on the Skolem-Herbrand-Gödel Theorem.

Ground resolution depends on checking that complementary ground literals in clauses match. In a similar way, general resolution depends on ensuring that complementary literals in two clauses can be unified. It is more efficient than ground resolution because unification makes forming ground clauses unnecessary.

Unification finds a most general unifier of sets of pairs of literals. The instance of the literals by their most general unifier is their most common general instance: every matching ground instance of two literals is an instance of it. An algorithm using transformations similar to those for first-order unification appears in Herbrand's thesis [88]. Another unification algorithm was given by Robinson [166], who showed that unifiability is decidable and that there always exists a most general unifier of sets of pairs of unifiable terms.

The resolution principle extends the ground resolution principle as follows:

> *Choose any two clauses C_1 and C_2 in a set of clauses. Rename the variables in these clauses uniquely and so that no variable occurs in both clauses. Choose $D \subseteq C_1$ and $E \subseteq C_2$ such that each literal D_i in D is a negative literal, each literal E_j in E is a positive literal, and there is a most general unifier θ of all of the atomic parts of the D_i and all of the E_j. Add the resolvent $((C_1 - D) \cup (C_2 - E))\theta$ to the set of clauses.*

Robinson [166] proved the following theorem.

Theorem 1.6 *(Resolution Theorem.) A formula in clause form is unsatisfiable if and only if $\square$ is produced as a resolvent after a finite number of applications of the resolution principle.*

If a set of clauses is satisfiable, resolution may not terminate and may keep producing new clauses.

Using the clause form of F', choose the clauses $\{\neg q(a, w)\}$ and $\{q(x, g(x)), q(x, h(x))\}$. A most general unifier of the atom $q(x, g(x))$, and the atomic part of $\neg q(a, w)$ is is $\theta = \{a/x,\ g(a)/w\}$. The resolvent of the clauses is $\{q(a, h(a))\}$. Resolving this clause with

$\{\neg q(a,w)\}$ yields the empty clause. It follows from the Resolution Theorem that the negation of F is a valid formula, that is

$$\exists x \forall y \forall z \neg[(\neg q(x,y) \equiv \exists t[\neg \forall y(q(x,y) \supset \neg p(f(v)))]) \wedge \forall x \neg q(v,x)]$$

is a valid formula.

There are usually many ways to apply the resolution principle. We could have chosen the clause $\{q(x,g(x)), q(x,h(x))\}$ and formed the resolvents $\{q(a,h(a))\}$ and $\{q(a,g(a))\}$ by resolution with the clause $\{\neg q(a,w)\}$. The clause $\{\neg q(x,g(x)), \neg q(x,z), \neg p(f(a))\}$ and the resolvent $\{q(a,h(a))\}$ have resolvent $\{\neg q(a,g(a)), \neg p(f(a))\}$, which when resolved with the clause $\{q(a,g(a))\}$ yields $\{\neg p(f(a))\}$. The empty clause is the resolvent of this clause, and the resolvent of $\{\neg q(a,w)\}$ and $\{q(x,g(x)), p(f(a))\}$.

The resolution principle alone, although much more efficient than methods based directly on ground clauses, can still be inefficient because of the necessity to convert formulas to Skolem normal form, and because redundant resolvents can be produced. In general, propositional resolution has non-polynomial time complexity [80]. The length of a resolution proof for infinitely many disjunctive normal form propositional tautologies which are sets of "pigeonhole clauses" is not bounded by any polynomial function of the length of the tautology[2]. An extended form of resolution gives polynomial length refutations for such propositional tautologies. Urquhart [191] refined this result by using similar techniques on "graph-based" sets of clauses. Unlike pigeonhole clauses, they have polynomial length proofs in an axiomatic system for the propositional calculus, and each clause C_n contains three literals instead of n literals. More generally, Baaz and Leitsch [13] have shown that the length of a shortest resolution proof for first-order clauses may be non-elementary in the length of a shortest proof in a classical logical calculus.

Proof methods based on variations of resolution have been developed which reduce redundancy by extending resolution for languages with equality [167], or by specifying which clauses should be chosen and which literals in those clauses should be tested for unifiability [124]. The variations maintain the property that $\square$ is produced as a resolvent if and only if the negation of the original formula in clause form is unsatisfiable.

One of these is linear resolution [123, 126] which reduces redundancy by always using the previous resolvent as one of the clauses used to form the next resolvent. SL-resolution, introduced by Kowalski and Kuehner [117], is a modification of linear resolution in which just one literal per clause is selected for testing unifiability.

1.2 Logic Programming

Apart from its use for formalizing and automating mathematical proofs, logic is also used as a programming language. The idea to use logic in this way is stated to have occurred in 1972 during a visit by R. Kowalski from the University of Edinburgh to the artificial intelligence group at Université d'Aix Marseilles [34], although the idea had occurred earlier to P. Hayes [116].

[2]The proof of this result does not use clauses, but it can easily be corrected.

The form of logic programming principally considered used Horn clauses[3]. A Horn clause program is a formula each of whose clauses contain one positive literal. To show that a closed formula $\neg G$ of the form $\neg\forall x_1 \cdots \forall x_n.\neg(A_1 \wedge \cdots \wedge A_n)$, where the A_i are positive literals, is a logical consequence of a program P, a form of resolution called *SL-resolution* was used to show that $P \wedge G$ is unsatisfiable.

The formula G in clause form is called a goal clause and it is a clause consisting of negative literals. Clauses in a Horn clause program are sometimes called definite clauses, and SL-resolution for definite clauses is called SLD-resolution[4] [11].

In SLD-resolution, a goal clause of the form

$$\{\neg A_1, \ldots, \neg A_{i-1}, \neg A_i, \neg A_{i+1}, \ldots, \neg A_n\}$$

and a definite clause from P of the form $\{A, \neg B_1, \ldots, \neg B_k\}$ whose variables have been renamed so that they are distinct from those in the goal clause and where $k \geq 0$, are used to form the resolvent or goal clause

$$\{\neg A_1\theta, \ldots, \neg A_{i-1}\theta, \neg B_1\theta, \ldots, \neg B_k\theta, \neg A_{i+1}\theta, \ldots, \neg A_n\theta\}$$

where θ is a most general unifier of the *selected literal* A_i, and A.

An example of Horn clause program is the formula

$$\begin{array}{l}\{\{reverse(nil, nil)\},\\ \{reverse(cons(X, Y), Z,), \neg reverse(Y, R),\\ \neg append(R, cons(X, nil), Z)\},\\ \{append(nil, X, X)\},\\ \{append(cons(X, Y), Z, cons(X, T)), \neg append(Y, Z, T)\}\}\end{array}$$

In first-order classical logic, a definite clause of the form

$$\{A_1, \neg A_2, \ldots, \neg A_n\}$$

is equivalent to the implication $A_n \wedge \cdots \wedge A_2 \supset A_1$. In logic programming, such an implication is abbreviated to $A_1 :- A_2, \ldots, A_n$. Using this abbreviation, the previous program can be written

```
reverse(nil, nil).
reverse(cons(X, Y), Z) :- reverse(Y, R),
                          append(R, cons(X, nil), Z).

append(nil, X, X).
append(cons(X, Y), Z, cons(X, T)) :- append(Y, Z, T).
```

A Horn clause program written in this abbreviated form is called a logic program. Similarly, a goal clause of the form $\{\neg G_1, \ldots, \neg G_n\}$ is abbreviated to $:- G_1, \ldots, G_n$.

[3]Horn originally defined similar clauses consisting of negated equations and at most one non-negated equation [95].

[4]However, G.A. Ringwood [165] states that SLD-resolution is D. Kuehner's [119] SNL-resolution without factoring, and that its identification [116] with SL-resolution is widely held but of questionable historical accuracy.

1.2.1 Least Model Semantics

Since the denotations of constant symbols in every Herbrand interpretation for a formula are fixed, it is usual to characterize a Herbrand interpretation by the set of all ground positive literals for which it is a model. We shall call the characterization of a Herbrand interpretation which is a model for a formula, a *Herbrand model* of the formula.

A basic fact about Horn clauses [95] is that the intersection of two Herbrand models for a Horn clause formula is a Herbrand model for the formula. The intersection of all Herbrand models of a formula is a unique model called the *least Herbrand model.*

Unique models are necessary for Horn clause logic to be regarded as a programming language. The identification of the least Herbrand model for this purpose is supported by the following result [52].

Theorem 1.7 *A ground positive literal is in the least Herbrand model of a logic program if and only if it is a logical consequence of the program.*

The existence of the least Herbrand model depends on the fact that only Horn clauses are being considered. Makowsky [128] shows that if such generic models are required, then first-order Horn clause logic with equality is the most general form of first-order logic with this property.

If a clause were to contain a disjunction $A \vee B$ where A and B are positive literals, there may be no least Herbrand model. There could be a Herbrand model which contains A but not B, and vice-versa. Their intersection would contain neither A nor B, and it would no longer be a Herbrand model for $A \vee B$. Clauses with such disjunctions arise in automated theorem proving problems. The lack of a least Herbrand model is not critical in this setting. The aim there is to show the unsatisfiability of a set of clauses rather than to investigate their properties as a programming language.

1.2.2 Refutations and Answers

Logic programming also differs from resolution based theorem proving because usually all refutations are of interest not merely the existence of a refutation.

This is because answers can be extracted from their composition of all most general unifiers computed for each refutation using a goal G and a logic program. The composition is an answer substitution θ, and $\neg G\theta$ is a logical consequence of P.

For example, using the previous program, the goal

```
:- reverse(cons(1, cons(2, nil)), X)
```

results in a refutation whose answer substitution is a most general unifier of the disagreement set:

$$\{\langle reverse(cons(1, cons(2, nil)), t), reverse(cons(x, y), z)\rangle$$
$$\langle reverse(y, r), reverse(cons(x_1, y_1), z_1)\rangle$$
$$\langle reverse(y_1, r_1), reverse(nil, nil)\rangle$$
$$\langle append(r_1, cons(x_1, nil), z_1), append(nil, x_2, x_2)\rangle$$
$$\langle append(r, cons(x, nil), z), append(cons(x_3, y_2), z_2, cons(x_3, u))\rangle$$
$$\langle append(y_2, z_2, u), append(nil, x_4, x_4)\rangle\}$$

which has the component $\langle X, cons(2, cons(1, nil))\rangle$.

Some logic programs also can generate several answers to the same goal. For example the goal

```
:- append(X, Y, cons(1, cons(2, cons(3, nil)))).
```

will produce four answers corresponding to the four possible values for `X` and `Y` such that `Y` appended to `X` is `cons(1, cons(2, cons(3, nil)))`. This immediately leads us to ask what can be computed using logic programs.

1.2.3 Computational Adequacy

A binary definite clause has the form $A :- B$, where A and B are atomic formulas. Tärnlund [189] has shown that every Turing computable function is computable by a logic program consisting of binary definite clauses[5]. Turing's Thesis [6] states that every algorithm can be programmed on a one-tape Turing machine.

If this is accepted, it follows that every effectively computable function can be computed by logic programs with binary clauses. This observation is also necessary for Horn clauses to be regarded as a programming language.

Many high-level programming languages provide abstractions such as modules, procedures, abstract data types, and polymorphism. Logic programs do not support most of these features directly because only first-order terms can be used as data structures. While such features do not change the computational adequacy of a programming language, they make programs more concise and easier to write and to understand.

1.2.4 Negation

Another feature not directly supported by logic programs is negation. It is often useful to find which relations do not hold [32]. Since Horn clauses are a subset of first-order logic, negation must be approximated.

The negation as failure rule is one such approximation. The resolution search space for a program P and a goal clause G can be organized as a tree called a *SLD-tree*. The goal clause is the root of the tree, and after having fixed a selected literal in a node of the tree, the children of that node are all goal clauses which can be formed by applying SLD--resolution. A SLD-tree has a *success branch* if it has a finite branch whose last resolvent is the empty clause. A SLD-tree is *finitely failed* if all of its branches are finite, and none is a success branch. A SLD-tree is a *fair SLD-tree* if in each infinite branch of the tree, an instantiated form of any literal in any node is the selected literal of a descendant node which has been formed after a finite number of applications of SLD-resolution.

If P is a Horn clause logic program, and G is a goal, the rule allows us to infer G from $comp(P)$ if and only if there is a finitely failed SLD-tree [11] for $P \cup \{G\}$, where $comp(P)$ is a formula produced from P.

[5] Lloyd [122], for example, surveys various characterizations of the computational adequacy of logic programming.

[6] Davis [44] gives an historical account of Turing's Thesis.