On treating negation within XSB (and upon extending XSB programming with a form of logical negation, and its relations to existing varieties of logic programming)

Jay Halcomb, Adam Pease

Teknowledge Corporation
1810 Embarcadero Road
Palo Alto, CA 94303, USA
jhalcomb@teknowledge.com
apease@teknowledge.com

"Observe carefully the small facts upon which the largest inferences depend." – A. Conan Doyle

Abstract

A variety of techniques within logic programming have developed for treating negated formulae. These essentially involve constructibility restrictions which are imposed upon complexity classes of quantificational formulae, and which thereby determine models and model classes of computer programs declaratively regarded. These constructibility restrictions are themselves implemented by imposing restrictions upon both the syntax and semantics of particular computing languages such as Prolog. Current alternative paradigms of logic programming in Prolog include the theorem-proving implementing semantics of first-order conjunctive normal form assertions, and definite and Horn clause logic programming systems. In this paper we discuss some relationships between the resolution principle and these systems, particularly as they concern implementations of negation -- from negation-as-failure through the treatment of negation within the Prolog extension XSB, involving tabling. We examine another proposal [T. Van Le, 1993] for the treatment of negation in Prolog, and we produce an interim report on implementing this form of negation within XSB, including a formal characterization, timing results, and comments upon heuristics. We also provide examples of the use of negation in XSB on knowledge based reasoning applications., and discuss practical performance issues as well as implications for knowledge engineering that follow from implementing this facility.

Introduction

Many problems in logical inference benefit from the use of expressive logical representations. In previous work, (Cohen et al, 1999) (Pease et al, 2000) we have shown the value in creating reusable logical representations. Our research and application efforts are now driving us to examine how we can achieve high performance, including speed of inference, on the reusable and expressive representations we create. Some of our current projects build on the XSB [Sagonas, et al, 2000] deductive database. XSB has several virtues which include its impressive speed in performing deductions in large knowledge bases which may have long inference chains. As we discuss below, XSB Prolog is, however, a rather basic language for knowledge representation compared to languages such as KIF [Genesereth, 1994] or CycL [Cycorp, 2001]. One feature needed in an expressive logical representation, in order to create a reusable formulation, is that of negation. While any predicate defined by a knowledge engineer could also have a negated version, that basic although inelegant solution does not scale well when one considers the need

to negate compound formulae. For this reason, we believe that logical negation is a key feature for a reasoning system that is to handle a knowledge base of any significant size. The issues to be considered in fully implementing logical negation in a knowledge representation system include: inferential soundness and completeness, and efficiency.

Background to the negation problem in logic programming: Unification and the Resolution Principle

J.A. Robinson [1965] introduced the **resolution principle** for logic programming. Various refinements and extensions of the general method have been formulated since. .Some version of the resolution principle is used in most modern logic programming languages, and in Prolog (and XSB) in particular. The resolution principle is an inference rule which maps a pair of sentences within a first-order language to another sentence, according to a schema which we will shortly produce: But to discuss the resolution principle, first we must mention the operations of substitution and unification.

Terms in our language are either: variables, individual constants, or the result of applying atomic function constants to variables or individual constants (as in, e.g., f(X,a,h(g(Y))). **Unification** is an operation upon two atomic formulae (say, A and B) which, when successful, produces a **substitution** of (possibly new) terms for those occurring in A and B such that the result of the substitution makes A and B **identical.** Such a substitution is called a unifier for A and B.

An important fact about the unifiers for two expressions A and B is that when they exist, there is a 'least' such unifier – one which makes the fewest substitutions possible which render A and B identical. Such a unifier is called a 'most general unifier'.

(Resolution principle, one form)

To explain the resolution principle, first we write Sub(**X**,E) to indicate the result of uniformly making substitutions of the vector of terms, **X**, for selected terms occurring in E. A convenient way of indicating such substitutions is with, e.g., the notation: [X/a, f(X)/b]..

From A1 v...v An and B1 v...v Bm, infer

Sub(**X**, (A1 v...v Aj-1 v Aj+1 v...v B1 v...v Bk-1 v Bk+1 v...v Bm), where **X** is that substitution of terms for terms which is the *most general unifier* for the expressions Aj and Bk; i.e., **X** is the least substitution of terms such that Sub(**X**,Aj) = Sub(**X**,not Bk).

The resolution principle itself relies upon the nontrivial notions of substitution of terms, and the unification of terms (and, by extension) of formulae. In consideration of space, though, we must omit discussion of these details. The important fact for us now is that resolution is a powerful generalization of the inference rule traditionally known to logicians as Modus Ponens. Using resolution, operating under appropriate selection/generation strategies for the materials to resolve upon in constructing a proof, it is possible to derive **refutation-complete** proof procedures for significant classes of expressions of FOL Refutation-completeness means that if a set of sentences of FOL is unsatisfiable (has no model), then the resolution process will derive, in finitely many steps, a contradiction from that set. Since a standard strategy for proving that a set of sentences, S, logically implies another sentence A is to conjoin the negation of A with the conjunction of those in S and then to infer a contradiction from that expression, the resolution principle can (after appropriate Skolemization and canonicalization) be used to show the logical implication of A from S, for various forms of sentences. [See, for instance, Chang and Lee, 1973]

Normal Forms

The second important fact we notice is that among the classes of formula types of FOL, for

logic programming purposes, the following are of particular note (in increasing order of restriction): the **conjunctive normal form**, the **Horn form**, and the **definite clause form**. We briefly discuss each of these.

The **conjunctive normal form** of a sentence of FOL is any **conjunction** of expressions, each conjunct of which is of the form (for some n and some m, possibly one or the other 0):

not A1 v ... v not An v An+1 v ... v Am

The important fact here is that we may legitimately speak of the conjunctive normal form of *any* expression of FOL, since any expression of FOL has a provably equivalent (in a certain sense) conjunctive normal form.

A **Horn clause** is any **conjunction** of expressions each of the form:

not A1 v ... v not An v Am

where either m or n may be 0.

A **definite clause** is a further restriction of the Horn form. It is any **conjunction** of expressions each of the form:

not A1 v ... v not An v An+1

where n may be 0.

By imposing restrictions or other refinements upon the general resolution principle, and by devising appropriate control (e.g., selection) strategies, the logic programmer can devise refutation-complete proof methods for each of the classes of normal forms we've mentioned.

Logic programs and theorem proving

Prolog is one instance of a logic programming language embodying Kowalski's precept 'algorithm=logic+control'. But 'vanilla' Prolog (Prolog without any 'not' operator at all) requires that the 'logic' of this equation be definite -clause logic: the restriction to FOL inference which results from evaluating only definite clauses, i.e., expressions of the form:

not P1 v ... v not Pn v Pn+1

where each Pi is an atomic literal formula. Such formulae are themselves logically equivalent to expressions of the form

P1 & ... & Pn-1 => Pn (and are written in Prolog as **Pn:-P1,...,Pn-1.**)

We notice that the restriction to definite clauses is imposed by the requirement that there be **at least one** positive (un-negated) literal in the disjunctions of the conjunctive normal form. Since Prolog simpliciter also doesn't allow for the evaluation of expressions of the form

P1 & ... & Pn => false (in Prolog this would be an illegal expression: **fail:- P1, ... , Pn.**)

this imposes a further reduction upon the canonical forms to which Prolog inference can be directly applied. Such clauses must be **definite** -- must have at least one positive atomic disjunct. This restriction means that direct Prolog resolution cannot directly represent nor resolve disjunctive facts of the form **not P1 v...v not Pn.**

Even when Prolog is augmented with the addition of its usual **'not'** operator, which expresses negation-as-failure, it is inferentially complete only expressions which have Horn clause canonical form.

For ground formula, the **'negation-as-failure'** method of evaluation amounts to the **'closed-world'** assumption, that a ground formula p(a) is false whenever either the assertion p(a) does not appear in the program, or it is not deducible from assertions in the program context.

This anomaly arises from the typical Prolog implementation of 'not', which behaves functionally as though:

not(X):-call(X),!,fail.
not(X):-true.

A typical sentence of FOL which has no equivalent Horn clause normal form (hence, still less any definite clause form) is the axiom expressing atomicity within Boolean algebras:

exists(X) all(Y)(not (X=0) & (x*y=y => x=y v y =0))

Without the means to resolve upon such expressions, the full inferential capability of FOL cannot be directly employed upon knowledge representations. On the other hand, it is well known that the introduction of such means introduces considerable complexity, and a concomitant inefficiency, into the evaluation of knowledge representations. An inference capability which operates to prove theorems in FOL which can be expressed in conjunctive normal form (and hence any theorem of FOL), is called a theorem prover. Theorem provers improve upon other forms of logic programming, like Prolog, by being able to resolve the full generality of expressions in conjunctive normal form, not merely those in Horn clause or definite clause form.

A closer look at the problem of negation in Prolog

Prolog with the addition of the usual '**not**' operator treats the evaluation of positive and negated goals differently. While a positive goal, **f(X),** is evaluated by searching for a proof of **exists(X) (f(X)),** a negated goal, **not(f(X)),** is evaluated for a proof of **all(X)(not f(X)),** as will be shown.

Logically, a sentence of the form, **all(X) all(Y) (b(X)=>a(Y))** is equivalent to **all(X) exists(Y)(b(Y) => a(X))** so that the Prolog evaluation of a goal **a(X):-b(Y)** ought to be evaluated as: **all(Y)(a(X) :- exists(Y) b(Y)).** And so it is, for *non-negated* predicates.

But in particular the evaluation of a negated body subgoal, **not b(Y),**
as in (*) **all(X) all(Y)(a(X):-not b(Y))** should be **all(X) (a(X) :- exists(Y)(not b(Y)).** However, standard Prolog evaluates rules of the form (*) as: **all(X) (a(X):- not exists((Y) b(Y)).**

Such inference is sound, when it succeeds for ground X, but it is not logically complete in the general case, for unbound X. This is demonstrated by Prolog's not returning a binding for X in such cases.

Prolog may also sometimes involve itself in infinitely looping processes during the resolution of certain forms of expression (those involving circular reference). This is because it employs a 'depth-first' search strategy, which may lead to it continue pursuing an unfruitful inference path. XSB provides a mechanism, tabling, which can control this behavior in many cases. [Warren, 2000].

Van Le negation: the 'non' operator

Van Le (*Techniques of Prolog Programming*, [1993]) offers another approach to the negation problem. In constructing an expert system shell, he defines a Prolog predicate 'non' such that the evaluation of (the attempt to unify) a goal **non(f(X))** seeks a maximally general substitution for f(X) such that f(X) *fails. I.e., it fails if there are no values of X such that f(X) holds, but moreover if there is a value for X such that non(f(X)) holds , that value is returned.* . The algorithm which implements this evaluation is not overly complex and Van Le is able to demonstrate for this method of treating negation that non(non(p(X)) implies p(X).

It is a feature of this method that such evaluation requires that Prolog has beforehand distinguished the function and constant symbols in use. It is because of this distinguished treatment that 'non'

is able to return values. That it does return such values yields another feature of the 'non' predicate: it can be used to define the universal quantifier 'all', (roughly expressed -- we omit details about renaming of variables to avoid clashes) through the definition:

all(X, p(X)):- not(non(p(X)).

The last clause may be read: it is false that **exists(X, non(p(X)).**

Similarly, **all(X, not p(X))** may be defined as **not(p(X)).** With suitable definitional extensions, it is then possible to define multiple quantification, as in all(X, exists(Y, p(X,Y)).

Examples in XSB

Employing an adaptation of Van Le's methodology, the following (and similar) data and query have been run at Teknowledge in XSB. The input materials were derived from the High Performance Knowledge Base project (Pease, et al, 2000).

Facts in the KB:

instance_of(hYP_TerritorialDispute_28532600, territorialDispute).

subset_of(territorialDispute, internationalConflict). isa(iraq, country).

instanceOf(iran, country).

opponentsInConflict(iran, iraq, hYP_TerritorialDispute_28532600)

Rules in the KB:

opponentsInConflict(Y, X, EVENT) :- opponentsInConflict(X, Y, EVENT).

capableOfDoing(unitedNationsOrganization, CONFLICT, mediators) :- instance_of(CONFLICT, internationalConflict), instance_of(COUNTRYA, country), instance_of(COUNTRYB, country), opponentsInConflict(COUNTRYA, COUNTRYB, CONFLICT).

instance_of(OBJ, SUPERSET) :- OBJ\=SUPERSET, instance_of(OBJ, SUBSET), subset_of(SUBSET, SUPERSET).

Input query: setof(WHO, all(CONFLICT, implies(and(instance_of(CONFLICT, territorialDispute), opponentsInConflict(iran, iraq, CONFLICT)), capableOfDoing(WHO, CONFLICT, mediators))), LIST).

Answer: [unitedNationsOrganization].

Although this query was run interpreted (and run over a miniscule KB), the evaluation of this query in XSB was so rapid (on a moderately quick PC), that the attempt to time it fell beneath the horizon of XSB's cpu timing mechanism, which measures cpu speed in 10,000ths of a second. However, compilation always produces increases in efficiency over interpretation.

Theorem provers, again

One theorem prover which has been implemented as an Prolog compiler is **PTTP**: the Prolog Technology Theorem Prover [Stickel, 1988, 1989]. PTTP is a refutation-complete inference engine for FOL which uses a slight extension of the Resolution Principle we have mentioned. One of the techniques which PTTP uses to achieve its level of generality is to employ negation, not as an operator, but in the form of individually negated predicates, i.e., for each predicate **p(X),** the negation of the predicate is expressed as a predicate, **not_p(X).** This translation is accomplished during the compilation state of a query (and so adds little to the execution process, while it is transparent to

the user), but it has the drawback that PTTP must compile each contained disjunction of an expression in conjunctive normal form in both positive and in several contrapositive forms. This last feature does impede execution. It also means that PTTP must sometimes be resolving over highly 'unnatural' forms; hence, forms less likely to produce fruitful inferences in the course of proof construction.

At Teknowledge we are presently implementing an XSB compiler for our knowledge representations which employs an adaptation of the Van Le strategy of negation.

References

Cohen, Chaudhri, Pease and Schrag . 1999. Does Prior Knowledge Facilitate the Development of Knowledge Based Systems, proceedings of AAAI-99.

Cycorp, 1999, Features of the CycL Language. http://www.cyc.com/cycl.html

Chang , C-L and Lee, R C-T. 1973. Symbolic Logic and Mechanical Theorem Proving., Academic Press, NewYork.

Genesereth, M. and Fikes, R. 1995, Knowledge Interchange Format v. 3.0. http://logic.stanford.edu/kif/Hypertext/kif-manual.html

Hogger, Christopher. 1984. Introduction to Logic Programming . Academic Press, London.

Pease, A., Chaudhri, V., Lehmann, F., and Farquhar, A., 2000, Practical Knowledge Representation and the DARPA High Performance Knowledge Bases Project, proceedings of KR-2000.

Robinson, J.A. January, 1965. A Machine Oriented Logic Based on the Resolution Principle. J. ACM 12, pp. 23-41.

Sagonas, Swift, Warren, Freire, Rao, Dawson, Kife. April 4, 2000. The XSB System, Version 2.2, Vols. 1, 2, Programmer's Manual, http://xsb.sourceforge.net/.

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

Stickel, M.E. June, 1989. A Prolog technology theorem prover: a new exposition and implementation in Prolog. Technical Note 464, Artificial Intelligence Center, SRI International, Menlo Park, California.

Van Le, T. 1993. Techniques of Prolog Programming (with implementation of logical negation and quantified goals). John Wiley & Sons, Inc., New York, 601 pp.

Warren, David S. July 31, 1999. Programming in Tabled Prolog (Draft). http://www.cs.sunysb.edu/~warren/xsbbook/book.html