Constructive Semantics for Negation

Clifford Walinsky

Technical Report No. CS/E 87-009

September, 1987

CONSTRUCTIVE SEMANTICS FOR NEGATION

Clifford Walinsky

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 87-009
September, 1987

Technical Report CS/E 87-009   September 1987

Constructive Semantics for Negation

Clifford Walinsky†

Department of Computer Science & Engineering
Oregon Graduate Center
Beaverton, OR 97006-1999
(503)690-1121

Abstract This paper examines implementation of negation in logic programs, distinct from the usual negation as failure. This implementation has three components: (i) 3-valued logic, (ii) unskolemized generalized definite clauses, and (iii).a generalized SLD-resolution procedure. The resulting declarative and procedural semantics have constructive qualities. When generalized definite clauses are used to implement iff-programs, analogous to those produced by the Clark completion procedure, the deductive mechanism is sound and weakly complete.

† Supported by United States Air Force Office of Scientific Research grant no. AFOSR-87-0064.

Constructive Semantics for Negation

1. Introduction

Definite Horn clause programs possess desirable constructive qualities. The declarative semantics of such programs are based on the least fixedpoint of a monotone operator (often referred to as T_P) [EK76]. The procedural semantics, utilizing SLD-resolution [EK76], dictate that upon successful execution of a query Q, an answer substitution σ will be returned, so that $Q\sigma$ is a valid consequence of the program.

When negation is introduced, the constructive characteristics are often lost, particularly for negation as failure, the most prevalent implementation of negation. Under this implementation, the declarative semantics of negated queries are derived from a subset of the greatest fixedpoint of T_P [AE82]. Also, this semantic operator, T_P, is no longer monotone. Using SLD-resolution, augmented with negation as failure, a query $\neg Q$ may fail even if $\neg Q\sigma$ is a valid consequence of the program, for some substitution σ.

The approach explored in this paper retains the constructive aspects of definite clause programs without negation. This approach utilizes *generalized* definite clause (*gdc*) programs. These programs permit explicit statement of negative facts. The negative facts are utilized to derive answer substitutions. For example, a predicate p with domain {0,...,5} may be true only for value 0. The program must then indicate that for values 1 through 5 predicate p is not true. The negative information is included within the program in the same form as the positive information. The program could appear as follows:

Example 1

```
 p(0).
~p(1).
  .
  .
  .
~p(5).
```

A deductive procedure for gdc programs, described in Section 5, can then deduce that p is true for value 0, and also that p is false for values 1,...,5. Furthermore, the query to find all values X for which p is false, ~p(X), succeeds with answer substitutions [i/X], where $1 \leq i \leq 5$.

In gdc programs it is possible to under-define predicates, resulting in incomplete definitions, and to create conflicting definitions, resulting in inconsistency. These problems arise from non-coordination between negative and positive definitions of a predicate. A certain class of programs provides the necessary coordination. Each program in this class is associated with an *iff-program*. Iff-programs are similar to programs produced by the Clark completion procedure [C78]. For an iff-program, an associated gdc program can be generated and used to define the declarative and procedural semantics of the original iff-program. The utility of generating gdc programs from iff-programs is mentioned in [JLM86].

It is not intended to entirely supplant negation as failure. Programs performing database applications typically contain a few predicates defined by many unconditional assertions. The number of clauses necessary to define the negation of such predicates is usually much larger than the number of clauses actually provided. Negation as failure can play a useful role here, if implemented soundly. Negation as failure can be explicitly invoked with a *fail* operator from within gdc programs. Thus a definition such as `~q(X₁, . . . ,Xₙ) :- fail q(X₁, . . . ,Xₙ)` defines `~q` by failure. Gabbay and Sergot proposed a similar idea in relation to their proposal for implementing negation [GS86]. In this way, constructive negation can be used in conjunction with other forms of negation, such as negation as failure.

Section 2 of this paper will present basic notation. Section 3 presents the declarative semantics of gdc programs. Section 4 describes iff-programs, and shows how to generate gdc programs from them. Section 5 describes the deductive mechanism for gdc programs associated with iff-programs. Finally, Section 6 discusses related work and future directions.

2. Syntax and Basic Concepts

2.1. Formulas, Clauses and Programs

In this section we describe the syntactic structure of definite clause programs with constructive extensions, referred to here as *generalized definite clause* (gdc) programs. *Atoms* are of the form $p(t_1, \dots, t_n)$, where p is a predicate symbol, and the t_i are finite terms. An n-tuple of terms is written as $\vec{t}$. A negative atom is of the form A. A *literal* is either an atom or a negative atom.

A *formula* is either a literal, or has the structure described below, where F and G are themselves formulas:

Conjunction:	$F \bigwedge G$
Disjunction:	$F \bigvee G$
Implication:	$F \rightarrow G$
Existential quantification:	$\exists x\, F$
Universal quantification:	$\forall x\, F$

Throughout, F and G will be used to denote formulas. To eliminate the possibility of conflicts between names of bound variables, assume that names used for quantified variables are not duplicated in any other formula.

A *clause* is of the form L :- F, where L, the *head* of the clause, is a literal, and F, the *body* of the clause, is a formula. A gdc program is a finite set of clauses. Every clause must be *closed*: each variable occurring in the body of a clause must be bound by a quantifier, or must occur in the head of the clause. All clauses of the form $p(\vec{t})$:- F or $p(\vec{t})$:- F *define* predicate p. No program can include clauses defining the distinguished predicate *true.*

2.2. Substitutions

The standard notation for substitutions, $[t_1/x_1, \dots, t_n/x_n]$, indicates that every occurrence of a variable x_i is to be replaced by term t_i. For this notation to correspond to a substitution, all x_i must be distinct variables, and all variables occurring within the t_i must be disjoint from $\{x_1, \dots, x_n\}$. Usually, σ, τ and η denote substitutions.

Composition of substitutions is defined by: $t(\sigma \circ \tau) = (t\ \tau)\sigma$. The distinguished substitution ι is the identity for composition.

Based on composition of substitutions, a partial order can be defined: $\sigma \subseteq \tau$ if there is a substitution ρ such that $\rho \circ \sigma = \tau$. For example, `[f(X)/Y]` $\subseteq$ `[f(a)/Y, b/Z]`. When $\sigma \subseteq \tau$, σ is *more general* than τ. A partial operation $\cup$ on substitutions is defined as follows: $\sigma \cup \tau$ is the most general substitution η such that $\sigma \subseteq \eta$ and $\tau \subseteq \eta$. The substitution returned by the $\cup$ operation is not always uniquely defined. But, every pair of substitutions returned by the $\cup$ operation will be of the form $[t_1/x_1, \ldots, t_n/x_n]$ and $[t_1'/x_1, \ldots, t_n'/x_n]$, and there will be a renaming ρ such that $t_i\rho = t_i'$. So we consider the result of the $\cup$ operation to be unique modulo renaming. When there is a substitution $\eta = \sigma \cup \tau$, σ and τ *unify*. The following illustrate the $\cup$ operation:

Example 2

`[a/X]` $\cup$ `[b/Y]` = `[a/X, b/Y]`

`[f(a,U)/X]` $\cup$ `[f(V,b)/X]` = `[f(a,b)/X, b/U, a/V]`

`[a/X]` and `[b/X]` do not unify

3. Declarative Semantics

3.1. Three-Valued Logic

The declarative semantics of gdc programs relies on 3-valued logic [F85]. The *truth values* of this logic are **t**, **f**, and **u**. Intuitively, these values are ascribed the meanings "true", "false" and "undefined", respectively. There is a partial ordering on the truth values: $\mathbf{u} \leq x$, and $x \leq x$, for all truth values x.

The truth values are combined with Boolean operators to form Boolean expressions. A Boolean expression e' is a *2-valued instance* of a Boolean expression e if every occurrence of **u** in e is replaced by either **t** or **f**. The valuation of a 2-valued instance is obtained by using the classical definitions of the Boolean operators. The valuation of any Boolean expression e is the greatest lower bound, with respect to the $\leq$ ordering on truth values, of the values of all 2-valued instances of e.

For example, $\mathbf{u} \wedge \mathbf{t}$ has 2-valued instances $\mathbf{t} \wedge \mathbf{t}$ and $\mathbf{f} \wedge \mathbf{t}$, with values **t** and **f**, respectively. The greatest lower bound is **u**. The full truth table for implication follows:

			y	
	$x \rightarrow y$	**t**	**u**	**f**
	t	**t**	**u**	**f**
x	**u**	**t**	**u**	**u**
	f	**t**	**t**	**t**

3.2. Interpretations and Models

For first-order logic programs, a domain of discourse is usually fixed for each program. The *Herbrand Base* of a program P is the set of all ground terms constructed from function symbols and constants occurring in program P. The *Herbrand Universe* of a program P is the set of ground atoms $p(\vec{T})$ constructed from predicates p occurring in program P and from terms of the Herbrand Base of P. $HB(P)$ and $HU(P)$ denote the Herbrand Base and Herbrand Universe, respectively, of a program P.

A (Herbrand) ***interpretation*** of a program P is a total mapping from $HU(P)$ into the truth values. Every interpretation must contain the map $true \mapsto \mathbf{t}$. The *minimal* interpretation of a program P maps $A \mapsto \mathbf{u}$ for all atoms $A \in HU(P)-\{true\}$. An interpretation I of a program P is *total* if $I[A] \neq \mathbf{u}$ for all $A \in HU(P)$.

Every interpretation can be extended to map over ground formulas and ground clauses, as follows:

$$I[A] = \neg I[A]$$

$$I[F \bigwedge G] = I[F] \bigwedge I[G]$$

$$I[F \bigvee G] = I[F] \bigvee I[G]$$

$$I[F \rightarrow G] = I[F] \rightarrow I[G]$$

$$I[\exists x\, F] = \begin{cases} \mathbf{t} \text{ if } I(F[t/x]) = \mathbf{t} \text{ for some } t \in HB(P) \\ \mathbf{f} \text{ if } I(F[t/x]) = \mathbf{f} \text{ for all } t \in HB(P) \\ \mathbf{u} \text{ otherwise} \end{cases}$$

$$I[\forall x\, F] = \begin{cases} \mathbf{t} \text{ if } I(F[t/x]) = \mathbf{t} \text{ for all } t \in HB(P) \\ \mathbf{f} \text{ if } I(F[t/x]) = \mathbf{f} \text{ for some } t \in HB(P) \\ \mathbf{u} \text{ otherwise} \end{cases}$$

$$I[L \text{ :- } F] = I[F] \rightarrow I[L]$$

All variables occurring free in formulas and clauses are universally quantified. So, if α is a non-ground formula or clause with free variables $x_1, \ldots, x_n$, then $I[\alpha] = I[\forall x_1 \cdots x_n(\alpha)]$. Finally, a program is interpreted as the conjunction of its clauses:

$$I[\{C_1, \ldots, C_n\}] = I[C_1] \bigwedge \cdots \bigwedge I[C_n]$$

Set notation is used to denote interpretations. If interpretation I is denoted by the set $\{L_1, \ldots, L_n\}$, then, for all atoms $A \in HU(P)$:

$$I[A] = \begin{cases} \mathbf{t} \text{ if } A = L_i \text{ for some } 1 \leq i \leq n \\ \mathbf{f} \text{ if } A = L_i \text{ for some } 1 \leq i \leq n \\ \mathbf{u} \text{ otherwise.} \end{cases}$$

While every interpretation has a set denotation, it is easy to construct sets that correspond to no interpretation. One such set is {p, ~p, true}. With this notation, the minimal interpretation is just {true}.

As an example of an interpretation, let $I = \{$~p, ~q, true$\}$. Then I[p] $= I$[q] $=$ **f**, and I[r] $=$ **u**. The following equalities hold for this interpretation:

$$\begin{aligned} I[\text{p:-}{}^{\sim}\text{q}\wedge\text{r}] &= I[{}^{\sim}\text{q}\wedge\text{r}] \rightarrow I[\text{p}] \\ &= (\neg I[\text{q}] \wedge I[\text{r}]) \rightarrow I[\text{p}] \\ &= (\mathbf{t} \wedge \mathbf{u}) \rightarrow \mathbf{t} \\ &= \mathbf{t}. \end{aligned}$$

The set notation for interpretations can be further extended to set operators:

$L \in I$ iff $I[L] = \mathbf{t}$

$I \subseteq J$ iff, for all ground literals $L \in I$, $L \in J$.

When S is a collection of interpretations, $L \in \cap S$ iff $L \in I$ for all $I \in S$.

As a consequence of the truth tables for the Boolean operators, the following monotone properties for interpretations hold:

Monotone Interpretations:

(i) For all formulas F, and interpretations I and J, $I[F] \leq J[F]$ when $I \subseteq J$.

(ii) For all interpretations I, formulas F, and substitutions σ and τ, $I[F\sigma] \leq I[F\tau]$ if $\sigma \subseteq \tau$.

These monotone results contrast sharply with 2-valued interpretations of logic programs with negated atoms in the bodies of clauses. Interpretations for such programs exhibit non-monotone behavior.

For an interpretation M and program P, M is a *model* of P iff $M[P] \neq \mathbf{f}$. This corresponds to the *weak models* of Lassez and Maher [LM85]. Note that the minimal interpretation $\{true\}$ is a model for every program. M is a *total* model for program P if M is a model of P and M is total.

3.3. Semantic Operator

T_P is an operator used to describe a class of models of a program P; it is similar to the semantic operator described by van Emden and Kowalski [EK76]. In this case, T_P is a *partial* mapping between interpretations. $T_P(I)$ is undefined when A :- F and A :- G are ground instances of clauses in P, and $I[F] = I[G] = \mathbf{t}$. Otherwise, $T_P(I)$ is defined, and for all ground literals L, $L \in T_P(I)$ whenever L :- F is a ground instance of a clause in P, and $I[F] = \mathbf{t}$. A *fixedpoint* of T_P is an interpretation I such that $T_P(I) = I$.

Lemma: (Fixedpoints are models) Every fixedpoint of T_P is a model of program P.‡

‡ Proofs of all original results stated in this paper are provided in [W87].

Hence, the collection of fixedpoints of T_P forms a sub-class of the collection of models for program P. Each model M in this class is *supported* [ABW85], in that if $M[L] = \mathbf{t}$, where L is a ground literal, there must be a ground instance L :- F of a clause and $M[F] = \mathbf{t}$.

Lemma (Fixedpoint Intersection) Let *FXP* be the collection of all fixedpoints of T_P. Then $\bigcap FXP \in FXP$.

This lemma parallels the model intersection property [EK76]. Since $\bigcap FXP$ defines a unique interpretation (which must be a model, by the above lemma), let *lfp* denote this unique *least fixedpoint.*

While *lfp* provides a non-effective characterization of the least fixedpoint, an effective characterization relies on iterations of T_P. Compute powers of T_P as follows:

$$T_P^0 = \{true\}$$

$$T_P^{n+1} = T_P[T_P^n] \text{ for all successor ordinals } n.$$

$$T_P^\lambda = \bigcup_{\alpha<\lambda} T_P^\alpha \text{ for all limit ordinals } \lambda.$$

The least ordinal λ for which $T_P[T_P^\lambda] = T_P^\lambda$ is the *closure ordinal*; let $ifp = T_P^\lambda$ for this λ, be the *iterated fixedpoint.* Because T_P is a partial mapping, it is possible that *ifp* will not be defined for a particular program. However, if *ifp* is defined, it will be a fixedpoint of T_P, hence a model of program P.

Lemma: If *ifp* is defined, *ifp* = *lfp*.

This lemma is demonstrated by showing that $ifp \subseteq M$, for all fixedpoints $M \in FXP$.

According to results of Fitting [F85], T_P is monotone and the poset of interpretations, ordered by set inclusion, is a complete semi-lattice. Hence, when *lfp* exists, *ifp* also exists. Due to the identity of the iterated and least fixedpoints, we can associate the closure ordinal of the iterated fixedpoint with the least fixedpoint.

There are two interesting cases for the least fixedpoint of T_P. The following program has undefined least fixedpoint:

Example 3

```
     p :- true.
    ~p :- true.
```

If a program contains no universal quantifiers, and the least fixedpoint is defined, it is guaranteed that the closure ordinal will be finite. This follows because all terms of the Herbrand Base must be finite. When universal quantifiers are present, however, the closure ordinal may be larger than ω. Consider the program below:

Example 4

```
    nat(0)  :- true.
    nat(s(N))  :- nat(N).

    r  :- ∀X nat(X).
```

In this example, ω iterations are needed to demonstrate that nat(X) is true for all X.

Hence, $\omega+1$ iterations are needed to produce a fixedpoint containing `r`.

The following lemma draws a connection between fixedpoints of T_P and 2-valued models for program P.

Lemma (*lfp* $\subseteq$ Total Models) For all total models M of a program P, *lfp*$\subseteq M$.

Consequently, when the least fixedpoint is undefined, P has no total models, and is inconsistent. The converse is not necessarily true. Consider the program below:

Example 5

```
 p :- ~p.
~p :- p.
```

This program has no total models. However, the least fixedpoint is defined, and is the minimal interpretation {`true`}.

4. Iff-Programs

With the ability to place negated atoms in the heads of clauses, and universal quantifiers within formulas, it is now possible to present statements of logical equivalence within programs. Logical equivalence provides a tight coordination between positive and negative definitions of a predicate.

Define an *iff-clause* to be of the form A -:- F, where A is an atom, and F is a formula. An *iff-program* is a set of iff-clauses, satisfying an additional syntactic requirement. If A -:- F is a clause in iff-program P, there can be no other clause B -:- G in P such that (variants of) A and B unify. It is intended that each iff-clause A -:- F completely defines atom A. So the presence of another iff-clause, with a possibly conflicting definition for A, is prevented. It is quite easy to construct a decision procedure to determine if this property is observed for all clauses of an iff-program.

4.1. Syntactic Negation

Associated with each iff-program is at least one gdc program, constructed using *syntactic negation*. Let *NEG* be a binary predicate over formulas. $NEG(F,F_{NEG})$ will hold when formula F_{NEG} is the syntactic negation of formula F. Usually, F_{NEG} denotes the fact that $NEG(F,F_{NEG})$ is true. To attain equivalence between semantic and syntactic negation, the *NEG* predicate is defined as follows:

For any atom A, $NEG(A,A)$ and $NEG(A,A)$ are both true.

For more complex formulas, the following statements for the *NEG* predicate are true:

$$NEG(F \wedge G, F_{NEG} \vee G_{NEG}).$$

$$NEG(F \vee G, F_{NEG} \wedge G_{NEG}).$$

$$NEG(F \wedge G, F \rightarrow G_{NEG}).$$

$$NEG(F \rightarrow G, F \wedge G_{NEG}).$$

$$NEG(\exists x\, F, \forall x\, F_{NEG}).$$

$NEG(\forall x\, F, \exists x\, F_{NEG})$.

For example, $NEG(\forall X\,[p(X,Y) \rightarrow q(X,Z)],\ \exists X\,[p(X,Y) \wedge {}^{\sim}q(X,Z)])$ holds.

The following commuting diagram relates syntactic and semantic interpretations of the NEG predicate:

$$\begin{tikzcd}
F \arrow[r, "NEG"] \arrow[d, "I"'] & F_{NEG} \arrow[d, "I"] \\
I[F] \arrow[r, "\neg"'] & \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}
\end{tikzcd}$$

Essentially, this diagram states that NEG is *homomorphic*: $\neg I[F] = I[F_{NEG}]$.

4.2. Generating GDC Programs from Iff-Programs

The NEG predicate can be used to construct a gdc program P_{GDC} associated with an iff-program P_{IFF}. P_{GDC} is the smallest set of generalized definite clauses such that A :- F and A :- F_{NEG} are contained in P_{GDC} whenever A -:- F is an iff-clause within P_{IFF}.

To illustrate this construction, consider the following iff-program:

Example 6

```
eq(a,a)  -:-  true.
eq(b,b)  -:-  true.
eq(a,b)  -:-  ~true.
eq(b,a)  -:-  ~true.

subsequence(nil,M)  -:-  true.
subsequence(X.L,nil)  -:-  ~true.
subsequence(X.L,Y.M)  -:-
      [eq(X,Y)  ∧  subsequence(L,M)]
      ∨ subsequence(X.L,M) .
```

This program defines a predicate subsequence(L,M), which is intended to be true when list L is a subsequence of list M. An empty list is represented by the term nil. A nonempty list is represented by the term $h.t$ (adopting the conventional infix notation for the function symbol .) where h, the head of the list, is an element from the domain {a,b}, and t, the tail of the list, is also a term representing a list. Notice that this program contains no quantifiers, so the closure ordinal of the least fixedpoint is finite. The construction above produces the following gdc program:

Example 7

```
(eq1)    eq(a,a)  :- true.
(eq2)    eq(b,b)  :- true.
(eq3)    eq(a,b)  :- ~true.
(eq4)    eq(b,a)  :- ~true.

(~eq1)  ~eq(a,a)  :- ~true.
(~eq2)  ~eq(b,b)  :- ~true.
(~eq3)  ~eq(a,b)  :- true.
(~eq4)  ~eq(b,a)  :- true.

(ss1)    subsequence(nil,M)  :- true.
(ss2)    subsequence(X.L,nil)  :- ~true.
(ss3)    subsequence(X.L,Y.M)  :-
              [eq(X,Y) ∧ subsequence(L,M)]
              ∨ subsequence(X.L,M) .

(~ss1)  ~subsequence(nil,M)  :- ~true.
(~ss2)  ~subsequence(X.L,nil)  :- true.
(~ss3)  ~subsequence(X.L,Y.M)  :-
              [~eq(X,Y) ∨ ~subsequence(L,M)]
              ∧ ~subsequence(X.L,M) .
```

With the Boolean equivalence operator $x \equiv y$ defined to be $(x \rightarrow y) \bigwedge (\neg x \rightarrow \neg y)$, it is clear that the declarative meaning of an iff-program P_{IFF} is identical to the declarative meaning of its associated gdc program P_{GDC}. In fact any model of P_{IFF} is a model for P_{GDC}.

The following lemma ensures that declarative semantics of all iff-programs are always defined.

Lemma (Fixedpoints of Iff-Programs Exist): If P is a gdc program associated with some iff-program, then *lfp* exists.

5. Procedural Semantics of Iff-Programs

The previous section has demonstrated construction of gdc programs from iff-programs. This section specifies the deductive mechanism necessary to execute such gdc programs. Generalized definite clause programs associated with a certain class of iff-programs render the execution procedure weakly complete and sound.

5.1. The SLDG Procedure

The procedure for executing a query on a program is called *SLDG* (*SLD* for *G*eneralized definite clause programs). It is derived from the SLD-resolution procedure. The procedure takes three input arguments: a program, a formula (or *query*), and a substitution. If the first argument is program P, it is usually omitted. Upon successful execution, the procedure *yields* a substitution. Like the original *SLD* procedure, *SLDG* is non-deterministic: for any combination of inputs, *SLDG* yields any one of a set of possible substitutions.

5.1.1. Execution of non-$\forall$ Formulas

The rules governing behavior of the *SLDG* procedure are simple extensions of the SLD-resolution rules. The following rules, based on the structure of the query formula, describe the substitutions that can be returned from successful execution. The case of the universal quantifier is deferred to Section 5.1.2, due to its complexity.

(*SLDG* true)
$SLDG(true,\sigma)$ yields σ.

(*SLDG* literal)
When L is a literal, $SLDG(L,\sigma)$ yields $SLDG(F',\mu')\circ\mu\circ\sigma$, where L':-F' is a variant of a clause in P such that all variables in the variant occur nowhere else, and $\mu\circ\sigma$ and μ' are the most general substitutions such that $L'\mu' = L(\mu\circ\sigma)$.

(*SLDG* $\bigwedge$)
$SLDG(F \bigwedge G,\sigma)$ yields $SLDG(F,\sigma)\cup SLDG(G,\sigma)$.

(*SLDG* $\bigvee$)
$SLDG(F \bigvee G,\sigma)$ yields $SLDG(F,\sigma)$ or $SLDG(G,\sigma)$.

(*SLDG* $\exists$)
$SLDG(\exists x\, F,\sigma)$ yields $SLDG(F,\sigma)$.

Notation used in these rules is somewhat abbreviated. When $SLDG(F,\sigma)$ yields an expression e containing a subexpression $SLDG(F',\tau)$, it is implied that further execution, through recursive invocation of the *SLDG* procedure, will yield a substitution η, which is combined, using the substitution operators, with the other substitutions in e.

Execution of the *SLDG* procedure assumes a fair computation rule [LM84] for selecting clauses for use by the (*SLDG* literal) rule, and for selecting sub-queries. This constitutes the search space of the procedure.

The *SLDG* procedure can fail to yield any substitutions. *SLDG finitely fails* if execution yields no substitutions, and the entire search space is finite. Also, *SLDG* can fail with an infinite search space; then *SLDG* does not terminate.

The above rules can be used directly to symbolically execute a query. The list of rules utilized to achieve successful termination is a *refutation.* Consider Example 7. The following steps lead to successful (symbolic) execution of the *SLDG* procedure for query `~subsequence(U,E.nil)` with the identity substitution:

SLDG(~subsequence(U,E.nil),ι)

Rule (*SLDG* literal), in merging with clause (~ss3), yields:
SLDG([~eq(X,Y) $\bigvee$ ~subsequence(L,M)]
$\bigwedge$ ~subsequence(X.L,M), [nil/M])
$\circ$ [X.L/U, Y/E]

Let σ = [nil/M], and τ = [X.L/U, Y/E].
Rule (*SLDG* $\bigwedge$) yields:
(*SLDG*(~subsequence(X.L,M), σ),
$\cup$ *SLDG*(~eq(X,Y) $\bigvee$ ~subsequence(L,M), σ))
$\circ\ \tau$

Rule (*SLDG* literal), in merging with clause (~ss2), yields:
(*SLDG*(true, σ)
$\cup$ *SLDG*(~eq(X,Y) $\bigvee$ ~subsequence(L,M), σ))
$\circ\ \tau$

Rule (*SLDG* true) yields:
(σ $\cup$ *SLDG*(~eq(X,Y) $\bigvee$ ~subsequence(L,M), σ))
$\circ\ \tau$

Rule (*SLDG* $\bigvee$) yields:
(σ $\cup$ *SLDG*(~subsequence(L,M), σ)) $\circ\ \tau$

Rule (*SLDG* literal), merging with a variant of clause (~ss2), yields:
[σ $\cup$ (*SLDG*(true, ι) $\circ$ [X'.L'/L] $\circ$ σ)] $\circ\ \tau$

Rule (*SLDG* true) yields:
[σ $\cup$ (ι $\circ$ [X'.L'/L] $\circ$ σ)] $\circ\ \tau$

This last substitution reduces to:
[X'.L'/L, nil/M, Y/E, X.X'.L'/U]

That is, execution of the *SLDG* procedure has demonstrated the theorem:

> Any list with more than one element is not a subsequence of a list with only one element.

Such universal statements demonstrate a heuristic usable in obtaining interesting, and possibly shorter, refutations:

> As much as possible, avoid instantiating variables to ground terms.

This heuristic is at variance with the commonly-used strategy to instantiate as many variables as possible, employed to obtain sound behavior of negation as failure in MU-Prolog [N85a], for example.

5.1.2. Execution of ∀ Formulas

With universal quantifiers in programs, T_P^{ω} may no longer be a fixedpoint, as demonstrated by Example 4. The progression to higher ordinals for the fixedpoint of T_P leads to incompleteness of the *SLDG* procedure. In many cases, however, *SLDG* can generate the actual valuation of a formula by *lfp*. The correct valuation is made possible by execution of *bounded* universally quantified formulas, of the form $\forall x(F \rightarrow G)$.

The algorithm used by the *SLDG* procedure for implementing the universal quantifier is presented first. This algorithm uses a difference operator on substitutions: $\sigma - x = [t/y \mid t/y \in \sigma \text{ and } y \neq x]$.

(*SLDG* $\forall$)

To compute the substitution returned by $SLDG(\forall x(F \rightarrow G), \sigma)$, perform the following steps:

(1) Let $\{\tau_1, \ldots, \tau_n\} = T$ be the full set of substitutions that $SLDG(F, \sigma)$ can yield. If, for some τ_i, there is no ground term t with $\tau_i \supseteq [t/x] \cup \sigma$, then halt with a *completeness-error.*

(2) Now construct a maximal unifying subset of T:

```
Select any $\tau_i \in T$.
Construct $S^{(i)} \subseteq T$ as follows:
    Let $S^{(i)}$ initially be $\{\tau_i\}$.
    For each $\tau_j \in T$ do:
            if $\tau_j - x$ unifies with $\tau_i - x$ then add $\tau_j$ to $S^{(i)}$.
$S^{(i)}$ must be a finite nonempty set.
```

(3) Since $S^{(i)}$ is a set of unifying substitutions, there must be a substitution that is common to all substitutions in $S^{(i)}$. Let $\tau^{(i)}$ be the most general substitution such that $\tau^{(i)} \supseteq \tau_j - x$, for all $\tau_j \in S^{(i)}$.

(4) For each $\tau_j \in S^{(i)}$, let $SLDG(G\,\tau_j, \iota)$ yield η_j. If there is a substitution $\eta = \bigcup_j \eta_j$, then $SLDG(\forall x(F \rightarrow G), \sigma)$ yields $\eta \circ \tau^{(i)}$.

To illustrate symbolic execution of a query with a universal quantifier, consider the program of Example 8 below, with the query $\forall$X[p(X,Y)→q(X,Z)], and the identity substitution:

Example 8

```
p(a,0)  :- true.
p(b,0)  :- true.
p(a,1)  :- true.

q(a,m)  :- true.
q(b,m)  :- true.
q(a,n)  :- true.
```

The following steps are taken by the *SLDG* procedure:

(1) Let $T = \{$[a/X,0/Y], [b/X,0/Y], [a/X,1/Y]$\}$.

(2) Let $S^{(1)} = \{$[a/X,0/Y], [b/X,0/Y]$\}$.

(3) Then $\tau^{(1)} =$ [0/Y].

(4) *SLDG*(q(X,Z)[a/X,0/Y], ι) yields η_1 = [m/Z], and *SLDG*(q(X,Z)[b/X,0/Y], ι) yields η_2 = [m/Z]. So $\eta_1 \cup \eta_2$ = [m/Z]. And finally, *SLDG*(∀X[p(X,Y)→q(X,Z)], ι) yields [0/Y,m/Z].

Note that for η_1 = [n/Z], η_1 does not unify with η_2, resulting in finite failure. The reader can verify that other possible substitutions returned for this query are [1/Y,m/Z] and [1/Y,n/Z].

The (*SLDG* ∀) rule introduces the possibility of a *completeness-error.* Similar to this error, the control-error is introduced in the MU-Prolog system [N85a] to indicate that a negated query cannot be grounded from execution of other queries. By contrast, the *SLDG* procedure signals completeness-error only when a universally quantified variable is not grounded following execution of the query in which it occurs. The localized nature of completeness-errors may permit easier detection and prevention than measures designed to eliminate control-errors.

5.2. Completeness and Soundness of SLDG

Weak-completeness and soundness results can be shown for a certain class of iff-programs. Such iff-programs are self-covering, and generate gdc programs with bounded universal quantifiers.

5.2.1. Self-Coverage and Well-Formed Programs

The final step of the Clark completion procedure requires addition of assertions of the form $\neg p(\overline{x})$, for all predicates p that are not defined in the program. The same concept for iff-programs can be used to achieve soundness and completeness of the deductive procedure.

Let P be an iff-program. P is *self-covering* if there is a ground instance A -:- F of an iff-clause in P, for every atom A in the Herbrand Universe of program P. Example 6 is not self-covering, since the eq predicate has no definitions for arguments that are lists, and the subsequence predicate has no definitions for arguments that are not lists. An enhancement, involving type-checking [M84], would admit programs that are not entirely self-covering, but self-covering within given type domains. Within certain type domains, Example 6 is self-covering.

Suppose P is a gdc program associated with an iff-program. It is possible that P contains universally quantified formulas that are not bounded; hence, they cannot be evaluated by the *SLDG* procedure. For example, because *NEG*(∃Xp(X), ∀X~p(X)) holds, the iff-clause q -:- ∃Xp(X) will generate a clause ~q :- ∀X~p(X) in the associated gdc program. This clause is not acceptable for interpretation by the *SLDG* procedure, since the universal quantifier is not bounded. A gdc program P is *well-formed* if each universally quantified formula in P is bounded. As the example above demonstrated, it is quite easy to construct iff-programs that have no associated well-formed gdc programs. However, it is possible to decide if an iff-program can generate a well-formed gdc program.

5.2.2. Completeness and Soundness Results

As demonstrated by Example 4, programs with universal quantifiers can possess infinite closure ordinals. This necessarily weakens any completeness result. Because each iteration of the semantic operator T_P is associated with a logical inference, requiring use of the (*SLDG* literal) rule, we can expect only weak-completeness: If $lfp[F\,\sigma] \neq \mathbf{u}$, it may be that $SLDG(F,\sigma)$ does not terminate.

Theorem (Weak-Completeness *SLDG*) Let $F\,\sigma$ be a ground formula, and P be a well-formed gdc program associated with some self-covering iff-program. If $SLDG(F,\sigma)$ terminates without a completeness-error, then $lfp[F\,\sigma] \neq \mathbf{u}$, and the following hold:

(i) If $lfp[F\,\sigma] = \mathbf{t}$, $SLDG(F,\sigma)$ is successful.

(ii) If $lfp[F\,\sigma] = \mathbf{f}$, $SLDG(F,\sigma)$ fails.

Note, as a consequence of this theorem, if $lfp[F\,\sigma] = \mathbf{u}$, then $SLDG(F,\sigma)$ is nonterminating. This is not the case if P is associated with an iff-program that is not self-covering. Proof of this theorem also relies on the capability of *SLDG* to signal completeness-error when universally quantified variables are not grounded.

The soundness result, below, relies on this weak-completeness result. This version of the soundness theorem must be augmented with an additional step, describing conditions in the event of finite failure.

Theorem: (Soundness of *SLDG*) Let P be a well-formed gdc program associated with a self-covering iff-program. Then the following hold:

(i) If $SLDG(F,\sigma)$ yields some substitution τ, then $lfp[F\,\tau] = \mathbf{t}$.

(ii) If $SLDG(F,\sigma)$ finitely fails, then $lfp[F\,\sigma] = \mathbf{f}$.

From the soundness theorem, a surprising observation emerges. The *SLDG* procedure may achieve a finite refutation, even though the closure ordinal is transfinite. In particular, consider the following iff-program, which determines when two sequences have a nonempty common subsequence:

Example 9

```
empty(nil)  -:- true.
empty(X.L)  -:- ~true.

common_subsequence(L,M)  -:-
      ∃X  [~empty(X)  ∧  subsequence(X,L)  ∧
             subsequence(X,M)] .
```

The associated gdc program includes the clause:

```
~common_subsequence(L,M)  :-
      ∀X  [(~empty(X)  ∧  subsequence(X,L))  →
             ~subsequence(X,M)] .
```

Consider evaluation of an atom `~common_subsequence(a.nil,b.nil)`. To reach a fixedpoint of T_P, at least ω iterations are required to obtain all valuations of `~empty(X) ∧ subsequence(X,a.nil)`, for all sequences `X`. However, only the substitution [`a.nil/X`] gives a valuation of **t** to this formula. All other sequences produce valuations of **f**. Weak completeness ensures that these other sequences will lead to failure, which may be finite. Hence, the refutation can be achieved in a finite number of steps.

6. Summary

6.1. Related Work

Much work has been done to characterize negation as failure. Shephardson [S84,S85] provides an extensive survey. Two approaches to a declarative semantics, the Closed World Assumption [R78] and the Clark program completion, are often at variance. However, Clark has demonstrated the soundness of negation as failure with respect to the completion of a program. Under sufficiently rigid conditions, these two approaches are identical. The class of models specified by the Clark completion is equivalent to the class specified by negation as failure only for "canonical" programs [JLM86]. Detection of such programs is undecidable, however. To overcome problems of non-monotone T_P, stratification schemes have been advanced [ABW85]. These schemes result in a least fixedpoint declarative semantics.

Fitting [F85] and Kunen [K86] present a 3-valued logic for definite clause programs in which the semantic operator T_P is monotone even for programs with negative literals. Using negation as failure, this permits completeness of the deduction mechanism for more programs than could achieved under 2-valued logic. Incompleteness still remains, however. As with the current presentation, this incompleteness is manifested in programs using either implicit or explicit universal quantification. This is not surprising, since introduction of negation permits description of complements of recursively enumerable sets, which may not be recursively enumerable.

Another proposal for deducing a negative query $\neg Q$, without resorting to failure, relies on a search up the proof tree for an ancestor query Q [PG86]. In addition, the original program must be expanded to include each contrapositive form of an original clause. For example, given the original clause `p :- ¬q,r`, the additional contrapositive clauses are `q :- ¬p,r` and `¬r :- ¬p,¬q`. Actual impact on performance of the new inference procedure and the additional clauses has yet to be revealed.

Another proposal would include with a program P a set of queries N which must not succeed [GS86]. The program is inconsistent if there is some query $Q \in N$ deducible from P. A negative query $\neg Q$ is deducible if $P \cup \{Q\}$ is inconsistent with N. While this computation rule is inexpensive to implement when queries do not contain quantified variables, the presence of quantified variables greatly increases the cost.

Universal quantification has not received as much attention as negation in logic programs. A universally quantified query $\forall x\, F$ can always be deduced by solving the equivalent query $\neg(\exists x\, \neg F)$ [LT84]. Under negation as failure, however, answer substitutions are discarded, so the mechanism is non-constructive.

6.2. Current Status & Future Work

The current implementation is written in C-Prolog [P85]. This has required both introduction of the negation operator , and a bounded universal quantifier. The negation operator is simply declared as a prefix operator, using Prolog's operator declaration facility. Hence, every clause of the form A :- F is actually a redefinition of a predicate named , with one argument, A. A query B then unifies with A if B unifies with A. This conforms with the (*SLDG* literal) rule.

The implementation of the universal quantifier is similar to that of the *all-solutions* predicates found in many Prolog implementations. It shares many of the deficiencies mentioned by Naish [N85b] for all-solutions predicates. In particular, all permutations of the set T must be generated, in order to generate complete sets of $S^{(i)}$. This eliminates the possibility of lazy evaluation for T, hence T must be finite. However, when the precondition of the bounded universal quantifier has no free variables, permutations are not needed, and lazy evaluation can be performed. Other possibilities may enhance the execution of the universal quantifier.

The following additional facilities also exist:

(1) Generation of gdc programs from iff-programs.

(2) Determination that an iff-program has non-conflicting definitions.

(3) Determination that an iff-program is self-covering, within a given set of type constraints.

The implementation requires facilities for information-hiding. This would allow the programmer's perception to remain at the level of iff-programs, rather than delve into the details of the associated gdc programs. This deficiency is manifested in the following ways: First, the programmer must have detailed knowledge of the implementation of the *NEG* predicate in order to ensure that gdc programs generated from iff-programs can execute efficiently. Also, debugging is performed on the underlying gdc program, rather than on the iff-program. These issues seem similar to those presented by compilers. While it is possible to view a compiled program as assembly language statements, greater coherence is achieved by creating tools, such as symbolic debuggers, that hide the actual implementation language.

References

[ABW85] Apt, Blair & Walker, "Towards a Theory of Declarative Knowledge," Technical Report, IBM Corp., Yorktown Heights, 1985.

[AE82] Apt & van Emden, "Contributions to the Theory of Logic Programming," *JACM*, 29(3), pp. 841-862, 1982.

[C78] Clark, "Negation as Failure", in *Logic and Data Bases*, Gallaire & Minker (eds.), Plenum Press, New York, pp. 293-322, 1978.

[EK76] van Emden & Kowalski, "The Semantics of Predicate Logic as a Programming Language," *JACM*, 23(4), pp. 733-742, 1976.

[F85] Fitting, "A Kripke-Kleene Semantics for Logic Programs," *J. of Logic Programming*, 4(1985), pp. 295-312.

[GS86] Gabbay & Sergot, "Negation as Inconsistency. I," *J. of Logic Programming*, 3(1), pp. 1-35, 1986.

[JLM86] Jaffar, Lassez & Maher, "Some Issues and Trends in the Semantics of Logic Programming," *1986 International Conference on Logic Programming Conference*, in Springer-Verlag Lecture Notes on Computer Science, vol. 225, pp. 223-241, 1986.

[K86] Kunen, "Negation in Logic Programming," submitted for publication to *J. of Logic Programming*, 1986.

[LM84] Lassez & Maher, "Closures and Fairness in the Semantics of Programming Logic," *Theoretical Computer Science*, 29(1984), pp. 167-184, 1984.

[LM85] Lassez & Maher, "Optimal Fixedpoints of Logic Programs," *Theoretical Computer Science*, 39(1985), pp. 15-25, 1985.

[LT84] Lloyd & Topor, "Making Prolog More Expressive," *J. of Logic Programming*, 1(2), pp. 225-240, 1984.

[M84] Mishra, "Towards a Theory of Types in Prolog," *Proc. IEEE Symposium on Logic Programming*, pp. 289-298, 1984.

[N85a] Naish, "Automating Control for Logic Programs," *J. of Logic Programming*, 2(1985), pp. 167-183.

[N85b] Naish, "All Solutions Predicates in Prolog," *Proc. IEEE Symposium on Logic Programming*, pp. 73-77, 1985.

[P85] Pereira, et al., *C-Prolog User's Manual*, edCAAD, Dept. of Architecture, University of Edinburgh, 1985.

[PG86] Poole & Goebel, "Gracefully Adding Negation and Disjunction to Prolog," *1986 International Conference on Logic Programming,* in Springer-Verlag Lecture Notes on Computer Science, vol. 225, 1986.

[R78] Reiter, "On Closed World Databases," in *Logic and Data Bases,* Gallaire & Minker (eds.), Plenum Press, New York, pp. 55-76, 1978.

[S84] Shepherdson, "Negation as Failure," *J. of Logic Programming,* 1(1984), pp. 51-79.

[S85] Shepherdson, "Negation as Failure. II," *J. of Logic Programming,* 3(1985), pp. 185-202.

[W87] Walinsky, "Constructive Semantics for Negation," CS/E TR-87/009, Oregon Graduate Center, 1987.