April 1986

Constructive negation in Horn-clause programs

Clifford Walinsky

Follow this and additional works at: http://digitalcommons.ohsu.edu/csetech

Recommended Citation

http://digitalcommons.ohsu.edu/csetech/142

This Article is brought to you for free and open access by OHSU Digital Commons. It has been accepted for inclusion in CSETech by an authorized administrator of OHSU Digital Commons. For more information, please contact champieu@ohsu.edu.
Constructive Negation in Horn-Clause Programs

Clifford Walinsky

Technical Report No. CS/E 86-003

April, 1986
Constructive Negation in Horn-Clause Programs

Clifford Walinsky
Oregon Graduate Center
Beaverton, OR 97006
(503) 690-1121

Abstract A natural mechanism for incorporating negation within Horn-clause programs is presented. Logic programs utilizing constructive negation provide negative as well as positive definitions of predicates. Constructive negation possesses several advantages over the negation mechanism used in standard Horn-clause programs: negation by failure. Important properties of constructive negation are presented. Though negative definitions can be provided by the programmer, a method for inference of negative definitions is described. The inference scheme also precludes production of inconsistent programs.
Constructive Negation

1. Introduction

In this paper I will investigate a sub-class of general logic programs, called extended Horn-clause programs. This sub-class is similar to standard Horn-clause programs in many respects. A key difference is the introduction of constructive negation. This mechanism for providing negation is very different from the usual implementation of negation in standard Horn-clause logic [S85]. However, a Horn-clause logic interpreter can execute programs containing constructive negation.

This paper is organized as follows: Section 2 introduces the syntax of extended Horn-clause programs. Sections 3 and 4 describe the declarative and procedural semantics of extended Horn-clause programs. Section 5 demonstrates that the declarative and procedural semantics are in fact equivalent by proving soundness and completeness. Section 6 compares constructive negation with the usual implementation of negation, negation by failure. Extended Horn-clause programs can be inconsistent. To prevent this, inference of a conservative extension is described in section 7. Section 8 describes opportunities for further research. Finally appendix A shows that standard Horn-clause interpreters can execute extended Horn-clause programs.

2. Syntax of Extended Horn-Clause Programs

In this section I will describe the syntactic structure of Horn-clause programs with constructive negation. Such programs will be referred to as extended Horn-clause programs. The syntax is very close to standard Horn-clause logic [CM81].

A term is either a variable, or a structure of the form \( f(t_1, \ldots, t_n) \), where each \( t_i \) is a term (when \( n = 0 \), the term is a constant). An atom is of the form \( p(t_1, \ldots, t_n) \), where \( p \) is a predicate name and each \( t_i \) is a term. A negative atom is expressed as \( \neg A \), where \( A \) is an atom. A literal is either an atom or a negative atom. Extended Horn-clause programs are composed of clauses. Each clause is of the form \( L \leftarrow \Phi \), where \( L \) is a literal and \( \Phi \) is a phrase. If \( \Phi_1 \) and \( \Phi_2 \) are phrases, \( z \) is a variable, and \( L \) is a literal, each phrase is of the form:
Constructive Negation

Conjunction: $\Phi_1 \land \Phi_2$
Disjunction: $\Phi_1 \lor \Phi_2$
Existential: $(\exists x) \Phi_1$
Literal: $L$

A variable $x$ is global within a clause $L \vdash \Phi$ if $x$ occurs within $L$. A variable $x$ is bound within the phrase $(\exists x)\Phi$. A clause $L \vdash \Phi$ is closed if every variable in $\Phi$ is either global or bound. Henceforth, all program clauses are assumed to be closed.

At this point, the notation of substitutions will be introduced. A substitution is a total function mapping expressions to expressions. Every substitution can be denoted by a set of pairs $[e_1/x_1, ..., e_n/x_n]$, where each $e_i$ is an expression, and all variables $x_i$ are distinct. Application of a substitution $\sigma$ to an expression is defined inductively by:

- $\sigma(x) = e$ if $\sigma$ contains a pair $e/x$; otherwise, $\sigma(x) = x$.
- $\sigma(a(e_1, ..., e_n)) = \sigma(a(e_1), ..., \sigma(e_n))$.

Composition of substitutions is defined by: $\sigma \cdot \tau(e) = \sigma(\tau(e))$. The empty substitution $\emptyset$ is the identity for composition. Expressions $e$ and $e'$ are unifiable with substitution $\sigma$ if $\sigma(e) = \sigma(e')$.

A difference operator is also defined: $\sigma - x = [e/y \mid e/y \in \sigma$ and $y \neq x]$. Substitutions cannot affect bound variables. If $Q$ is a quantifier, $\sigma((Qx)e) = (Qx)\sigma'(e)$, where $\sigma' = \sigma - x$. A substitution is a renaming if the substitution is of the form $[y_1/x_1, ..., y_n/x_n]$, where every $y_i$ is a variable.

$\sigma(L \vdash \Phi)$ is an instance of a clause $L \vdash \Phi$ if $\sigma = [t_1/x_1, ..., t_n/x_n]$, and each $x_i$ is a global variable. A ground expression contains no variables. $\sigma(L \vdash \Phi)$ is a ground instance of a clause $L \vdash \Phi$ if $\sigma(L \vdash \Phi)$ is a ground clause. $\sigma(L \vdash \Phi)$ is a variant of a clause $L \vdash \Phi$ if $\sigma$ is a renaming substitution.
3. Declarative Semantics of Extended Horn-Clause Programs

As with general first-order predicate logic, a model-theoretic meaning can be ascribed to each extended Horn-clause program. An interpretation is a set of ground literals. For the declarative semantics of extended Horn-clauses alone, every interpretation must contain the distinguished atom true. I will denote the fact that a phrase $\Phi$ is a valid consequence under interpretation $I$ by $I \models \Phi$. $I \models \Phi$ when $I \models \sigma(\Phi)$ for every ground instance $\sigma(\Phi)$ of $\Phi$. Given that $\Phi$ is a ground phrase, $\Phi$ is such that $I \models \Phi$ according to the following recursive definition:

- $\Phi$ is a literal, and $\Phi \in I$.
- $\Phi = \Phi_1 \land \Phi_2$, and $I \models \Phi_1$ and $I \models \Phi_2$.
- $\Phi = \Phi_1 \lor \Phi_2$, and $I \models \Phi_1$ or $I \models \Phi_2$.
- $\Phi = (\exists z)\Phi_1$, and there is a substitution $\sigma = [t/z]$, with $t$ a ground term, such that $I \models \sigma(\Phi_1)$.

An interpretation $I$ is true of a ground clause $L \vdash \Phi$ if $I \models L$ when $I \models \Phi$. Interpretation $I$ is true of a non-ground clause $L \vdash \Phi$ if $I$ is true of every ground instance of $L \vdash \Phi$. Finally, interpretation $I$ is true of a program $\Pi$ if $I$ is true of each clause in $\Pi$. An interpretation true of a program $\Pi$ is a model of $\Pi$.

For any program $\Pi$ there is a class of models of $\Pi$, $\Sigma(\Pi)$. A distinguished member of this class is the least model, $\cap \Sigma(\Pi)$, which is contained in all members of $\Sigma(\Pi)$. Existence and uniqueness of the least model is demonstrated below.

**Lemma 3.1:** For every program $\Pi$, the following is true:

(i) $\cap \Sigma(\Pi)$ is a model.

(ii) $\cap \Sigma(\Pi)$ is the smallest model of $\Pi$.

**Proof:**

(i) Say that $\cap \Sigma(\Pi)$ is not a model. There must be a ground instance $L \vdash \Phi$ of a clause in $\Pi$ such that $\cap \Sigma(\Pi) \models \Phi$ and $\cap \Sigma(\Pi) \not\models L$. But for all models $M$, if $M \models \Phi$, then $M \models L$. So $\cap \Sigma(\Pi)$
Constructive Negation

\[ \models L, \text{ and } \cap \Sigma(\Pi) \text{ must be a model.} \]

(ii) Say there is a model \( M' \) and ground literal \( L \) such that \( L \not\in M' \) and \( L \in \cap \Sigma(\Pi) \). There must be a ground instance \( L \models \Phi \) of a clause in \( \Pi \) such that \( \cap \Sigma(\Pi) \models \Phi \). So \( M \models \Phi \) for all models \( M \) and \( M \models L \). Hence \( M' \) is not a model. \( \Box \)

The definition of the least model provides a non-effective characterization. An equivalent definition places a structure on the least model. Define the least model to be the smallest interpretation satisfying the inductive criteria below:

\[ M_0(\Pi) = \{ \text{true} \}. \]

\[ L \in M_n(\Pi) \text{ if } L \models \Phi \text{ is a ground instance of a clause in } \Pi \text{ and } M_i(\Pi) \models \Phi \text{ for some } i < n. \]

Define \( M(\Pi) \) to be the limit of the \( M_n \): There is some finite number \( N \) for which \( M(\Pi) = M_N(\Pi) = M_{N+k}(\Pi) \) for all non-negative integers \( k \).

In a standard way, I will now show that \( M(\Pi) \) is equivalent to \( \cap \Sigma(\Pi) \).

Lemma 3.2: For every program \( \Pi \), \( \cap \Sigma(\Pi) = M(\Pi) \).

Proof: The proof is in two parts, first demonstrating \( M(\Pi) \subseteq \cap \Sigma(\Pi) \), then demonstrating \( \cap \Sigma(\Pi) \subseteq M(\Pi) \).

\( M(\Pi) \subseteq \cap \Sigma(\Pi) \): This inclusion is demonstrated by induction on the subsets \( M_i \) of \( M(\Pi) \).

(Basis) \( \text{true} \in M_0 \), and \( \text{true} \in M \) for all \( M \in \Sigma(\Pi) \).

(Induction) Assume \( M_i \subseteq \cap \Sigma(\Pi) \) for all \( i < n \). If \( L \in M_n \), there is a ground instance \( L \models \Phi \) of a clause in \( \Pi \) such that \( M_i \models \Phi \) for some \( i \). By the hypothesis \( M \models \Phi \) for all \( M \in \Sigma(\Pi) \). Since every \( M \in \Sigma(\Pi) \) is a true interpretation of \( \Pi \), \( L \in M \).

\( \cap \Sigma(\Pi) \subseteq M(\Pi) \): I now show that for all phrases \( \Phi \) such that \( M \models \Phi \), there is a minimum integer \( i \) such that \( M_i \models \Phi \).

(Basis) If \( \Phi = \text{true} \), then \( i = 0 \).

(Induction) Assume that for phrases \( \Phi_1 \) and \( \Phi_2 \), \( M_i \models \Phi_1 \) and \( M_j \models \Phi_2 \). Also assume that \( M \models \Phi \) for all \( M \in \Sigma(\Pi) \). Now examine each of the phrase forms:

[\( \Phi \text{ is a literal } L \) Since \( M \models \Phi \) for all models \( M \in \Sigma(\Pi) \), there must be a ground instance \( L \models \Phi_1 \)
Constructive Negation

of a clause in \( \Pi \) such that \( M \models \Phi_1 \) for all \( M \). Then \( M_i \models \Phi_1 \) by hypothesis, and \( M_{i+1} \models L \).

\( \Phi = \Phi_1 \land \Phi_2 \) Since \( M \models \Phi \) for all models \( M \in \Sigma(\Pi) \), \( M \models \Phi_1 \) and \( M \models \Phi_2 \). By hypothesis, \( M_i \models \Phi_1 \) and \( M_j \models \Phi_2 \). If \( i > j \), then \( M_i \models \Phi_2 \), so \( M_i \models \Phi_1 \land \Phi_2 \).

\( \Phi = \Phi_1 \lor \Phi_2 \) Since \( M \models \Phi \) for all models \( M \in \Sigma(\Pi) \), \( M \models \Phi_X \) for \( X \in \{1, 2\} \), and all \( M \). By hypothesis \( M_k \models \Phi_X \), for \( k \in \{i, j\} \). So \( M_k \models \Phi_1 \lor \Phi_2 \).

\( \Phi = (\exists \tau)\Phi_1 \) Since \( M \models \Phi \) for all models \( M \in \Sigma(\Pi) \), \( M \models \sigma(\Phi_1) \) for some \( \sigma = [\tau/x] \), and all \( M \). By hypothesis, \( M_i \models \sigma(\Phi_1) \), so \( M_i \models (\exists \tau)\Phi_1 \). ∎

Let \( \Pi \models \Phi \) denote the fact that phrase \( \Phi \) is a valid consequence of the least model, \( M(\Pi) \).

The following lemma will be used in the soundness and completeness proofs to come.

**Lemma 3.3:** If \( L \models \Phi \) is a clause in program \( \Pi \), and \( \Pi \models \tau(\Phi) \) for some substitution \( \tau \), then \( \Pi \models \tau(L) \).

Proof: For every substitution \( \eta \) such that \( \eta \cdot \tau(\Phi) \) is a ground instance, \( \Pi \models \eta \cdot \tau(\Phi) \). By definition of \( M(\Pi) \), \( \Pi \models \eta \cdot \tau(L). \) Since this is true of all substitutions \( \eta \), \( \Pi \models \tau(L) . \) ∎

4. Procedural Semantics of Extended Horn-Clause Programs

Having described the intended connection between logic programs and logic, in this section I will describe the actual execution of logic programs. The procedure for executing a standard Horn-clause logic program is called *SLD-resolution* [EK76], which is a variation of the resolution procedure [R65]. Programs are evaluated with a similar procedure that I will call SLD-resolution with constructive negation (or SLD-CN-resolution). SLD-CN can be described by an algorithm accepting three arguments: a program, a phrase, and a substitution. When initiated, SLD-CN is provided the empty substitution. Successful execution of the SLD-CN procedure returns a substitution. When unsuccessful, SLD-CN either terminates, returning \( \bot \), or does not terminate. I will denote the fact that SLD-CN returns substitution \( \tau \) when given a program \( \Pi \), a phrase \( \Phi \) and a substitution \( \sigma \) by \( \text{SLD-CN}(\Pi, \Phi, \sigma) = \tau \) (when program \( \Pi \) is understood, this argument is omitted). The computation procedure of SLD-CN is described by the following recursive
Constructive Negation

rules:

- \( \text{SLDCN}(\phi, \bot) = \bot. \)

- \( \text{SLDCN}(\text{true}, \sigma) = \sigma. \)

- \( \text{SLDCN}(\phi_1 \land \phi_2, \sigma) = \text{SLDCN}(\phi_1, \text{SLDCN}(\phi_2, \sigma)), \) where \( \{r, s\} = \{1, 2\}, \) and \( r \neq s. \)

- \( \text{SLDCN}(\phi_1 \lor \phi_2, \sigma) = r \) if \( \text{SLDCN}(\phi_r, \sigma) = r; \) otherwise \( \text{SLDCN}(\phi_s, \sigma) \) if \( \text{SLDCN}(\phi_r, \sigma) = \bot, \) again where \( \{r, s\} = \{1, 2\}, \) and \( r \neq s. \)

- \( \text{SLDCN}(\exists x \phi, \sigma) = \text{SLDCN}(\rho(\phi), \sigma) - y, \) where \( \rho = [y/x] \) and \( y \) is a variable not contained in \( \sigma \) or \( \phi. \)

- \( \text{SLDCN}(L, \sigma) = \text{SLDCN}(\phi, \rho \cdot \sigma) \) (\( L \) is a literal) if there is a variant \( L' :\phi \) of a clause in \( \Pi \) such that \( \rho(L') = \rho \cdot \sigma(L) \) and \( L' :\phi \) has no variables in common with \( \sigma \) or \( L; \) otherwise \( \bot. \)

This rule involves use of the unification algorithm [R65].

The values \( r \) and \( s \) used in these rules are not necessarily constant. Values of \( r \) and \( s \) can be any that will lead to successful termination of SLDCN, as if chosen by an oracle. In this paper I will assume the SLDCN procedure relies on such an oracle. By contrast, the Prolog language specifies the values for \( r \) and \( s \) to be constants 1 and 2, respectively. Consequently, execution of the Prolog interpreter may not terminate in cases when successful execution could be achieved.

To demonstrate this definition of SLDCN, assume that \( \Pi \) contains the clauses:
Constructive Negation

Example 4.1

(i) \( eq(a,a) :- true. \)
(ii) \( eq(b,b) :- true. \)
(iii) \( \neg eq(a,b) :- true. \)
(iv) \( \neg eq(b,a) :- true. \)
(v) \( subsequence(nil,M) :- true. \)
(vi) \( subsequence(cons(X,L), cons(Y,M)) :- \)
\[ \begin{align*}
& eq(X,Y) \land subsequence(L,M) \\
& \lor subsequence(cons(X,L),M).
\end{align*} \]
(vii) \( subsequence(cons(X,L),nil) :- \neg true. \)
(viii) \( \neg subsequence(cons(X,L),nil) :- true. \)
(ix) \( \neg subsequence(nil,M) :- \neg true. \)
(x) \( \neg subsequence(cons(X,L), cons(Y,M)) :- \)
\[ \begin{align*}
& \neg eq(X,Y) \lor \neg subsequence(L,M) \\
& \land \neg subsequence(cons(X,L),M).
\end{align*} \]

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 \( cons(h,t) \) 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. The following expressions are equivalent, providing a refutation of \( subsequence(U, cons(b,nil)) \)
Constructive Negation

SLDCN(subsequence (U, cons (b, nil)), \emptyset)

Unification with a variant of clause (vi):
= SLDCN([eq (X, Y) \land \text{subsequence}(L', M')] \\
\lor \text{subsequence}(\text{cons}(X, L'), M')),
\text{cons}(X, L')/U, b/Y, \text{nil}/M')

Use of the SLDCN rule for \lor:
= SLDCN([eq (X, Y) \land \text{subsequence}(L', M')],
\text{cons}(X, L')/U, b/Y, \text{nil}/M')

Use of the SLDCN rule for \land:
= SLDCN\text{subsequence}(L', M'), \text{SLDCN}(eq (X, Y),
\text{cons}(X, L')/U, b/Y, \text{nil}/M'))

Unification with a variant of clause (ii):
= SLDCN\text{subsequence}(L', M'), \text{SLDCN}(true,
\text{cons}(b, L')/U, b/X, b/Y, \text{nil}/M'))

Use of the SLDCN rule for true:
= SLDCN\text{subsequence}(L', M'),
\text{cons}(b, L')/U, b/X, b/Y, \text{nil}/M')

Unification with a variant of clause (v):
= SLDCN(true,
\text{cons}(b, \text{nil})/U, \text{nil}/L', M'/M'', b/X, b/Y, \text{nil}/M'))

Use of the SLDCN rule for true:
= \text{cons}(b, \text{nil})/U, \text{nil}/L', M'/M'', b/X, b/Y, \text{nil}/M')

Consequently, list cons (b, nil) is a subsequence of itself.

5. Soundness and Completeness of SLDCN

In discussing the soundness of the SLDCN procedure, I will need to define the length of a successful refutation. This length is defined to be the number of SLDCN rules invoked during the refutation. For example, the length of the refutation for Example 4.1 is 7. This will now be used to prove the following:

**Lemma 5.1:** SLDCN(\tau(\Phi), \sigma \cdot \tau = \eta \text{ when } SLDCN(\Phi, \sigma \cdot \tau) = \eta.

Proof: By induction on the length of a refutation.

(Basis) Assume the length is 1. Then \Phi = true, and SLDCN(\Phi, \sigma \cdot \tau) = \sigma \cdot \tau = SLDCN(\tau(\Phi), \sigma \cdot \tau).

(Induction) Assume the lemma is true for all refutations of length less than n, and a refutation
Constructive Negation

of phrase \( \Phi \) under substitution \( \sigma \cdot r \) is of length \( n \). Now consider each possible phrase form:

\([\Phi \text{ is a literal } L]\) There must be a variant \( L' \) of a clause such that \( \rho(L') = r \cdot \sigma \cdot \tau(L) \). Then

\( \text{SLDCN}(L, \sigma \cdot r) = \text{SLDCN}(\Phi', \rho \cdot \sigma \cdot r) \). The hypothesis holds, so

\( \text{SLDCN}(\Phi', \rho \cdot \sigma \cdot r) = \text{SLDCN}(\tau(\Phi'), \rho \cdot \sigma \cdot r) \).

Variables in \( \Phi' \) are distinct from \( r \), so

\( \text{SLDCN}(\tau(\Phi'), \rho \cdot \sigma \cdot r) = \text{SLDCN}(\Phi', \rho \cdot \sigma \cdot r) \).

Also \( \text{SLDCN}(\tau(L), \sigma \cdot r) = \text{SLDCN}(\Phi', \rho \cdot \sigma \cdot r) \).

\([\Phi = \Phi_1 \land \Phi_2]\) Then \( \text{SLDCN}(\Phi, \sigma \cdot r) = \text{SLDCN}(\Phi_1, \sigma \cdot r) \). The hypothesis holds, so

\( \text{SLDCN}(\Phi_1', \sigma \cdot r) = \text{SLDCN}(\tau(\Phi_1'), \sigma \cdot r) \), and:

\[ \text{SLDCN}(\Phi_1', \text{SLDCN}(\Phi_2'), \sigma \cdot r)) = \]

\[ \text{SLDCN}(\Phi_1', \text{SLDCN}(\tau(\Phi_2'), \sigma \cdot r)) = \]

\[ \text{SLDCN}(\tau(\Phi_1'), \text{SLDCN}(\tau(\Phi_2'), \sigma \cdot r)) = \]

\[ \text{SLDCN}(\tau(\Phi_1', \text{SLDCN}(\Phi_2'), \sigma \cdot r)) \).

\([\Phi = \Phi_1 \lor \Phi_2]\) Then \( \text{SLDCN}(\Phi, \sigma \cdot r) = \text{SLDCN}(\Phi_X, \sigma \cdot r) \), where \( X \in \{1, 2\} \). The hypothesis holds, so

\( \text{SLDCN}(\Phi_X, \sigma \cdot r) = \text{SLDCN}(\tau(\Phi_X), \sigma \cdot r) = \text{SLDCN}(\tau(\Phi_1 \lor \Phi_2), \sigma \cdot r) \).

\([\Phi = (\exists z)\Phi_1]\) Then \( \text{SLDCN}(\Phi, \sigma \cdot r) = \text{SLDCN}(\rho(\Phi_1), \sigma \cdot r) \cdot y \), where \( \rho = [y/z] \). The hypothesis holds, so

\( \text{SLDCN}(\rho(\Phi_1), \sigma \cdot r) \cdot y = \text{SLDCN}(\tau \cdot \rho(\Phi_1), \sigma \cdot r) \cdot y \). The variable \( y \) is not contained in \( r \), so

\( \text{SLDCN}(\tau \cdot \rho(\Phi_1), \sigma \cdot r) \cdot y = \text{SLDCN}(\tau(\exists z)\Phi_1), \sigma \cdot r) \).

Several corollaries are now easy to prove and will help in the soundness theorem to come:

**Corollary 5.2:** \( \text{SLDCN}(\Phi, \sigma) = \text{SLDCN}(\sigma(\Phi), \emptyset) \cdot \sigma \).

**Proof:** Let \( \sigma = \sigma \cdot \emptyset \).

**Corollary 5.3:** If \( \text{SLDCN}(\Phi, \sigma) = r \), then there exists a substitution \( \eta \) such that \( r = \eta \cdot \sigma \).

**Proof:** \( \text{SLDCN}(\Phi, \sigma) = \text{SLDCN}(\sigma(\Phi), \emptyset) \cdot \sigma \). Let \( \eta = \text{SLDCN}(\sigma(\Phi), \emptyset) \).

I am now ready to provide proof that execution of a logic program provides semantics identical to the intended logical interpretation of the program. This is the soundness property:

**Theorem 5.4:** \( \text{SLDCN} \) is sound in that \( \text{SLDCN}(\Phi, \sigma) = r \) implies that \( \Pi \models \tau(\Phi) \).
Constructive Negation

Proof: By induction on the length of a refutation.

(Basis) If the length is 1, \( \Phi = \text{true} \), \( \text{SLDCN}(\Phi, \sigma) = \sigma \), and \( \Pi \models \sigma(\text{true}) \).

(Induction) Assume the theorem is true for all refutations of length less than \( n \), and a refutation of phrase \( \Phi \) under substitution \( \sigma \) is of length \( n \). Now consider each possible phrase form:

[\( \Phi \) is a literal \( L \)] There must be a variant \( L' \) of a clause such that \( \rho(L') = \rho \cdot \sigma(L) \). Then \( \text{SLDCN}(L, \sigma) = \text{SLDCN}(\Phi', \rho \cdot \sigma) = \tau \). The hypothesis holds, so \( \Pi \models \tau(\Phi') \). By lemma 3.3, \( \Pi \models \tau(L') \). I now must show that \( \tau(L') = \tau(L) \). By corollary 4.4, there is a substitution \( \eta \) such that \( \tau = \eta \cdot \rho \cdot \sigma \). So \( \tau(L') = \eta \cdot \rho \cdot \sigma(L') \), and because \( \sigma \) contains no variables occurring in \( L' \), \( \eta \cdot \rho \cdot \sigma(L') = \eta \cdot \rho(L') \). Because \( \rho(L') = \rho \cdot \sigma(L) \), applying \( \eta \) to both sides gives \( \eta \cdot \rho(L') = \eta \cdot \rho \cdot \sigma(L) \). But \( \eta \cdot \rho \cdot \sigma = \tau \), so \( \tau(L') = \tau(L) \), and \( \Pi \models \tau(L) \).

[\( \Phi = \Phi_1 \land \Phi_2 \)] Then \( \text{SLDCN}(\Phi, \sigma) = \text{SLDCN}(\Phi_1, \sigma) \cdot \text{SLDCN}(\Phi_2, \sigma) = \tau \). The hypothesis holds, so \( \Pi \models \rho(\Phi) \) and \( \Pi \models \tau(\Phi) \), where \( \text{SLDCN}(\Phi_1, \sigma) = \rho \). For every substitution \( \eta \), \( \Pi \models \eta \cdot \rho(\Phi_1) \), and by corollary 5.3, at least one substitution \( \eta \) exists such that \( \eta \cdot \rho = \tau \). Therefore \( \Pi \models \tau(\Phi) \), so \( \Pi \models \tau(\Phi_1 \land \Phi_2) \).

[\( \Phi = \Phi_1 \lor \Phi_2 \)] Then \( \text{SLDCN}(\Phi, \sigma) = \text{SLDCN}(\Phi_1, \sigma) \cdot \text{SLDCN}(\Phi_2, \sigma) = \tau \), for \( X \in \{1, 2\} \). The hypothesis holds, so \( \Pi \models \tau(\Phi_X) \), and \( \Pi \models \tau(\Phi_1 \lor \Phi_2) \).

[\( \Phi = (\exists x)\Phi' \)] Then \( \text{SLDCN}(\Phi, \sigma) = \text{SLDCN}(\rho(\Phi'), \sigma) \cdot \rho(y/x) = \tau \), where \( \rho = [y/x] \). The hypothesis holds, so \( \Pi \models \tau \cdot \rho(\Phi') \). Since \( y \) is not contained in \( \tau \), \( \Pi \models \tau((\exists x)\Phi') \). \( \square \)

As a result of this theorem, a phrase \( \Phi \) is deducible from a program \( \Pi \) by SLDCN if there is a substitution \( \tau \) such that \( \text{SLDCN}(\Phi, \begin{array} \sigma \end{array}) = \tau \). \( \tau \) is also called a satisfying substitution for \( \Phi \).

To provide a completeness proof, I must first demonstrate that SLDCN is monotonic. Define a partial order on substitutions as follows: \( \sigma \subset \subset \tau \) if and only if there is a substitution \( \eta \) such that \( \sigma \cdot \eta = \tau \).

Lemma 5.5: If \( \text{SLDCN}(\Phi, \sigma) = \tau \) and \( \sigma' \subset \sigma \), then \( \text{SLDCN}(\Phi, \sigma') = \tau' \) for \( \tau' \subset \tau \).

Proof: By induction on the length of a refutation.

(Basis) If the length is 1, \( \Phi = \text{true} \). Then \( \text{SLDCN}(\text{true}, \sigma) = \sigma = \tau \). And when \( \sigma' \subset \sigma \),
Constructive Negation

SLDCN(true, σ') = σ' = τ'. So τ' ⊆ τ.

(Induction) Assume the lemma is true for all refutations of length less than n, and a refutation of phrase Φ under substitution σ is of length n. Now consider each possible phrase form:

[Φ is a literal L] There must be a variant L' of a clause in Π such that ρ(L') = ρ · σ(L) for some substitution ρ. Then SLDCN(L, σ) = SLDCN(Φ', ρ · σ) = τ. Let σ = η · σ'. Then SLDCN(L, σ') = SLDCN(Φ', ρ · η · σ') = τ, and τ' ⊆ τ.

[Φ = Φ₁ ∧ Φ₂] Then SLDCN(Φ₁ ∧ Φ₂, σ) = SLDCN(Φ₂', SLDCN(Φ₁, σ) = τ. Let SLDCN(Φ₁, σ) = ρ. The hypothesis holds, so SLDCN(Φ₁, σ') = ρ', and SLDCN(Φ₂, ρ') = τ', where ρ' ⊆ ρ and τ' ⊆ τ. Hence, SLDCN(Φ₁ ∧ Φ₂, σ') = τ', and τ' ⊆ τ.

[Φ = Φ₁ ∨ Φ₂] Then SLDCN(Φ₁ ∨ Φ₂, σ) = SLDCN(Φ₁, σ) = τ, where X ∈ {1, 2}. The hypothesis holds, so SLDCN(Φ₁, σ') = τ', where τ' ⊆ τ. SLDCN(Φ₂, σ') = τ', and τ' ⊆ τ.

[Φ = (∃z)Φ₁] Then SLDCN((∃z)Φ₁, σ) = SLDCN(ρ(Φ₁), σ) - y = τ, where ρ = [y/z]. The hypothesis holds, so SLDCN(ρ(Φ₁), σ') - y = τ', where τ' ⊆ τ. Therefore SLDCN((∃z)Φ₁, σ') = τ', and τ' ⊆ τ. □

As a corollary to monotonicity, the following is easily proved:

Corollary 5.6: If SLDCN(σ(Φ), ∅) = τ, then there is a substitution τ' such that SLDCN(Φ, ∅) = τ' and τ' ⊆ τ · σ.

Proof: By corollary 5.2, SLDCN(Φ, σ) = SLDCN(σ(Φ), ∅) · σ = τ · σ. And by lemma 5.5, since ∅ ⊆ σ, SLDCN(Φ, ∅) = τ, where τ' ⊆ τ · σ. □

Using corollary 5.6 now affords proof of completeness of the SLDCN procedure:

Theorem 5.7: For any ground phrase Φ such that Π ⊨ Φ, there is a successful refutation of Φ under substitution ∅.

Proof: By induction on the structure of the least model M(Π).

(Basis) true ∈ M(Π), and SLDCN(true, ∅) = ∅.

(Induction) Assume that for all ground phrases Φ₁ and Φ₂ such that Π ⊨ Φ₁ and Π ⊨ Φ₂, there are successful refutations of Φ₁ and Φ₂ under substitution ∅. Now consider a phrase Φ for
Constructive Negation

which $\Pi \models \Phi$, and consider each phrase form:

[\Phi\text{ is a ground literal } L] \text{ There must be a ground instance } \rho(L' \leftarrow \Phi') \text{ of a clause in } \Pi \text{ such that } \\
\rho(L') = L \text{ and } \rho(\Phi') = \Phi_1. \text{ Then } \text{SLDCN}(L, \emptyset) = \text{SLDCN}(\Phi', \rho). \text{ By corollary 5.2, } \text{SLDCN}(\Phi', \rho) = \text{SLDCN}(\rho(\Phi'), \emptyset) \cdot \rho. \text{ The hypothesis holds, and } \text{SLDCN}(\rho(\Phi'), \emptyset) \cdot \rho = \tau = \text{SLDCN}(L, \emptyset).

[\Phi = \Phi_1 \land \Phi_2] \text{ By the hypothesis, } \text{SLDCN}(\Phi_r, \emptyset) = \sigma \text{ and } \text{SLDCN}(\Phi_s, \emptyset) = \tau, \text{ where } \{r, s\} = \{1, 2\}. \text{ SLDCN}(\Phi_s, \text{SLDCN}(\Phi_r, \emptyset)) = \text{SLDCN}(\Phi_s, \sigma) = \text{SLDCN}(\sigma(\Phi_s), \emptyset) \cdot \sigma, \text{ by corollary 5.2. Since, } \\
\Phi_s \text{ is a ground phrase, } \sigma(\Phi_s) = \Phi_s \text{ and } \text{SLDCN}(\Phi_1 \land \Phi_2, \emptyset) = \tau \cdot \sigma.

[\Phi = \Phi_1 \lor \Phi_2] \text{ By the hypothesis, } \text{SLDCN}(\Phi_X, \emptyset) = \tau, \text{ where } \{X\} = \{1, 2\}. \text{ Then } \text{SLDCN}(\Phi_1 \lor \Phi_2, \emptyset) = \tau.

[\Phi = (\exists x)\Phi_1] \text{ Let } \Phi_1 = \sigma(\Phi'). \text{ By the hypothesis, } \text{SLDCN}(\sigma(\Phi'), \emptyset) = \tau. \text{ Then } \text{SLDCN}((\exists x)\Phi', \emptyset) = \text{SLDCN}(\rho(\Phi'), \emptyset) - y, \text{ where } \rho = [y/x]. \text{ By corollary 5.6, } \text{SLDCN}(\rho(\Phi'), \emptyset) - y = \tau', \text{ where } \\
\tau' \subseteq \tau \cdot \sigma. \text{ Hence, } \text{SLDCN}((\exists x)\Phi', \emptyset) = \tau'. \Box

6. Negation By Failure

Standard Horn-clause interpreters usually provide negation in a manner different from constructive negation. This section briefly describes their negation and compares it with constructive negation.

In model-theoretic terms, the negated atom $\neg A$ is a valid consequence of a program $\Pi$, if $\Pi \not\models A$; and $\Pi \not\models A$ if there is some ground instance $A'$ of $A$ such that $A' \not\in M(\Pi)$.

Due to the soundness and completeness of the SLDCN procedure, we could expect similar results for negated atoms: when \text{SLDCN}(\Pi, A, \emptyset) returns $\bot$, thus failing to find a satisfying substitution, then $\Pi \models \neg A$. This implementation of negation is referred to as negation by failure. I will denote the fact that a literal $L$ is to be deduced through negation by failure with \text{not } L. \text{ Hence } \text{SLDCN}(\Pi, \text{not } L, \sigma) \text{ returns } \sigma \text{ if } \text{SLDCN}(\Pi, L, \sigma) \text{ returns } \bot.

It has been found that soundness and (weak) completeness of negation by failure is obtained for standard SLD-resolution [AE82]. Unfortunately, these promising results do not clearly represent some of the real difficulties involved when using negation. First, there must be
Constructive Negation

some basis for deciding that \textit{not} \(L\) is deducible by negation by failure from program \(\Pi\) if a refutation for \(L\) does not terminate. In general this cannot be done. So the soundness and completeness results do not allow for non-termination. It is also not possible in general to determine \textit{a priori} that a given computation will not terminate. A second, and more serious, objection to the use of negation by failure is the new meaning imparted to variables occurring within a query \textit{not} \(L\).

By the soundness of SLDCN-resolution, when literal \(L\) is deducible by SLDCN with a satisfying substitution \(\sigma\), every ground instance of \(\sigma(L)\) is in the least model of program \(\Pi\). However, no such search is conducted in order to determine if \textit{not} \(L\) is deducible. In Example 4.1, deduction of query \textit{not} \textit{subsequence} \((L, \text{cons} (a, \text{nil}))\) fails, since SLDCN finds satisfying substitutions \([\text{nil}/L]\) and \([\text{cons} (a, \text{nil})/L]\) for query \textit{subsequence} \((L, \text{cons} (a, \text{nil}))\). The query \textit{not} \textit{subsequence} \((L, \text{cons} (a, \text{nil}))\) would not have failed had \(L\) been instantiated to \(\text{cons} (b, \text{nil})\), for example. We see for negation by failure SLDCN-resolution no longer conducts a search for a satisfying substitution.

One possible solution is to employ a \textit{safe} evaluation strategy. This strategy would delay evaluation of any \textit{unsafe} query \textit{not} \(L\) until \(L\) is a ground literal \([C78]\). Consequently, certain queries will \textit{flounder}: all possible refutations terminate in unsafe queries. Since it is not generally possible to syntactically detect when a query will flounder, syntactic limitations have been proposed for logic programs that guarantee all queries will not flounder \([S85]\). I feel these limitations are too severe for general-purpose programming tasks.

Even with a \textit{safe} evaluation strategy, negation by failure presents severe problems to development of reliable logic programs. To create reliable software and hardware, it is useful to think of distinct components as \textit{black boxes}. This principle can be carried over to Horn-clause programs: the content of each clause may be considered a black box, because we are interested only in the overall behavior of each predicate. The presence of negated atoms in conjunction with negation by failure semantics destroys the ability to consider the content of each clause as
Constructive Negation

a black box. Whenever a refutation involves negation by failure, the meaning of the variables occurring in the refutation is subtly altered, and the programmer must delve into the contents of clauses to discover where negation by failure arose.

Constructive negation serves as a usable alternative to negation by failure. The procedural semantics of negation are unified with the SLDCN procedure. Hence the "black box" principle is preserved. Further, the notion of floundered queries no longer arises. Several challenges remain for full utilization of constructive negation. First among these challenges is to determine if it is possible to provide negative definitions for large classes of programs. Second, a basis for eliminating logically-inconsistent programs must be established. Fortunately, one idea is sufficient to resolve both problems. This concept, due to David Maier in a private communication, involves extensions of programs.

7. Conservative Extension of Extended Horn-Clause Programs

As a principle of predicate logic, any literal is deducible from an inconsistent program \[E72\]. A program is inconsistent if both literals \(A\) and \(\neg A\) are deducible. This principle is relevant to extended Horn-clause programs. An interpretation \(I\) is inconsistent if both \(I \models A\) and \(I \models \neg A\) for some ground atom \(A\). A program \(\Pi\) is inconsistent if \(M(\Pi)\) is inconsistent. The following rule for construction of the least model of a program \(\Pi\) duplicates the results of predicate logic for inconsistent programs:

If \(A, \neg A \in M(\Pi)\), then \(L \in M(\Pi)\) for any ground literal \(L\) in the Herbrand base of \(\Pi\).

The Herbrand base of a program \(\Pi\) is the set of all ground literals constructed from predicate and function symbols occurring within \(\Pi\). Unfortunately, due to issues of undecidability, detection of inconsistent programs cannot generally be performed. But syntactic restrictions on programs eliminate the possibility of constructing syntactically correct inconsistent programs. Of course the problem with this approach is that consistent programs which do not conform to the syntactic criteria will not be accepted.
Constructive Negation

The necessary condition for consistency is as follows:

In a given program, there cannot be a pair of variants \( A \rightarrow \Phi \) and \( \neg A' \rightarrow \Phi' \) of clauses such that \( A \) and \( A' \) are unifiable.

If this condition is observed for a program \( \Pi \), there is no possibility of having literals \( A \) and \( \neg A \) within the least model of \( \Pi \). Hence, \( \Pi \) must be consistent.

Unfortunately, the necessary condition is in many cases too strong, eliminating too many programs. In many programs the necessary condition for consistency is violated in the following manner: Given a program with a clause defining a predicate \( p \), \( p(x) \rightarrow \Phi \), there will almost always be a dual clause: \( \neg p(x) \rightarrow \Phi' \). Frequently there is a definite symmetry between phrases \( \Phi \) and \( \Phi' \). To exploit this symmetry, the dual clauses of a program can be inferred so that consistency of the program is maintained.

I define a model \( M' \) to be an extension of a model \( M \) if \( M \subseteq M' \). Similarly, a program \( \Pi' \) is an extension of a program \( \Pi \) if \( M(\Pi) \subseteq M(\Pi') \). An extension \( \Pi' \) of a program \( \Pi \) is conservative if \( \Pi' \) is consistent whenever \( \Pi \) is consistent.

For notational convenience, define the total function \( \text{NOT} \) with one argument, a literal, returning a literal, according to the following rules:

- \( \text{NOT}(A) = \neg A \).
- \( \text{NOT}(\neg A) = A \).

where \( A \) is an atom.

Also, define a binary relation \( \text{NEG} \) on phrases. Denote by \( \text{NEG}(\Phi) = \Phi' \) the fact that \((\Phi, \Phi') \in \text{NEG}\). \( \text{NEG} \) is the largest relation for which the negation condition holds for all tuples in \( \text{NEG} \):

For all consistent interpretations \( I \) and phrases \( \Phi, I \models \Phi \) implies \( I \not\models \text{NEG}(\Phi) \).

Due to the maximality of \( \text{NEG} \), if \( I \) is consistent, and \( I \models \Phi \) implies \( I \not\models \Phi' \), then \( \text{NEG}(\Phi) = \Phi' \), and \( \text{NEG}(\Phi') = \Phi \).
Constructive Negation

To characterize the negation relation $\text{NEG}$, every tuple in $\text{NEG}$ must be an instance of one of the following rules:

(i) $\text{NEG}(L) = \text{NOT}(L)$.
(ii) $\text{NEG}(\Phi_1 \land \Phi_2) = \text{NEG}(\Phi_1) \lor \text{NEG}(\Phi_2)$.
(iii) $\text{NEG}(\Phi_1 \lor \Phi_2) = \text{NEG}(\Phi_1) \land \text{NEG}(\Phi_2)$.
(iv) $\text{NEG}(\exists x(\Phi_1 \land \Phi_2)) = (\forall x)(\Phi_1 \rightarrow \text{NEG}(\Phi_2))$.
(v) $\text{NEG}(\exists x(\Phi_1 \land \Phi_2)) = (\forall x)(\Phi_2 \rightarrow \text{NEG}(\Phi_1))$.
(vi) $\text{NEG}(\forall x(\Phi_1 \rightarrow \Phi_2)) = (\exists x)(\Phi_1 \land \text{NEG}(\Phi_2))$.

Note that these rules introduce universal quantification. A brief presentation of the declarative semantics of universal quantification is included here. Assume $(\forall x)(\Phi_1 \rightarrow \Phi_2)$ is a ground phrase, and $I$ is an interpretation.

Then $I \models (\forall x)(\Phi_1 \rightarrow \Phi_2)$ if and only if $I \models \sigma(\Phi_2)$ whenever $I \models \sigma(\Phi_1)$ for all substitutions $\sigma = [t/x]$.

With this brief description of universal quantification, I now provide proof that every tuple in $\text{NEG}$ satisfies the negation condition.

**Lemma 7.1**: If $\text{NEG}(\Phi) = \Phi'$ is an instance of a rule for the negation relation, then $I \models \Phi$ implies $I \not\models \Phi'$ for all consistent interpretations $I$.

**Proof**: The proof proceeds by induction on the structure of the $\text{NEG}$ relation.

(Basis) Consider a literal $L$. Assume $I \models L$. Since $I$ is consistent, $I \not\models \text{NOT}(L)$, so according to rule (i), $I \not\models \text{NEG}(L)$.

(Induction) Assume the negation condition is maintained for phrases $\Phi_1$ and $\Phi_2$.

$[\Phi = \Phi_1 \land \Phi_2]$ Assume $I \models \Phi$. Then $I \models \Phi_1$ and $I \models \Phi_2$. Also $I \models \text{NEG}(\Phi)$ if $I \models \text{NEG}(\Phi_1)$ or $I \models \text{NEG}(\Phi_2)$, according to rule (ii). By hypothesis, the negation condition holds, so $I \not\models \text{NEG}(\Phi_1)$ and $I \not\models \text{NEG}(\Phi_2)$. Hence, $I \not\models \text{NEG}(\Phi)$.

$[\Phi = \Phi_1 \lor \Phi_2]$ Assume $I \models \Phi$. Then $I \models \Phi_1$ or $I \models \Phi_2$. Also $I \models \text{NEG}(\Phi)$ if $I \models \text{NEG}(\Phi_1)$ and $I \models \text{NEG}(\Phi_2)$. Hence, $I \not\models \text{NEG}(\Phi)$.
Constructive Negation

NEG(Φ₂), according to rule (iii). By hypothesis, the negation condition holds, so I \not\models NEG(Φₓ) for X ∈ {1, 2}. Hence, I \not\models NEG(Φ).

[Φ = (∃z)(Φ₁ ∧ Φ₂)] Let NEG(Φ) = (∀z)(Φ₁ → Φ₂), according to rule (iv). Rule (v) can be handled similarly. Assume I \models Φ. Then, for some substitution σ = [t/z], I \models σ(Φ₁) and I \models σ(Φ₂).

By the hypothesis, I \models σ(Φ₂) implies I \not\models NEG(σ(Φ₂)). I \models (∀z)(Φ₁ → NEG(Φ₂)) if I \models NEG(σ(Φ₂)) for every σ = [t/z] for which I \models σ(Φ₁). We have I \models σ(Φ₁) but I \not\models NEG(σ(Φ₂)), so I \not\models NEG(Φ).

[Φ = (∀z)(Φ₁ → Φ₂)] According to rule (vi), NEG(Φ) = (∃z)(Φ₁ ∧ NEG(Φ₂)). Assume I \models Φ. Then I \models σ(Φ₂) whenever I \models σ(Φ₁), for all σ = [t/z]. By the hypothesis, if I \models σ(Φ₂), then I \not\models NEG(σ(Φ₂)). And if I \models σ(Φ₁) and I \not\models NEG(σ(Φ₂)), then I \not\models (∃z)(Φ₁ ∧ NEG(Φ₂)). Otherwise, I \not\models σ(Φ₁) for all substitutions σ. Again, I \not\models NEG(Φ).

For computation purposes, the NEG relation must be compressed into a function neg. This is achieved with the following rules for compression of a relation:

- If NEG(Φ) = Φ', NEG(Φ) = Φ", and σ(Φ') = Φ" for some substitution σ, then neg(Φ) = Φ'.
- Also if NEG(Φ) = Φ', NEG(Φ) = Φ", and Φ' and Φ" are not unifiable, then neg(Φ) = Φ' only. This case requires some method for deciding the appropriate compression.

An extension function can now be defined that builds dual definitions from clauses within a program. Let EXT be an extension function accepting a program Π and a compressed negation function neg, returning the smallest program Π' such that:

(i) All clauses of Π are contained in Π'.

(ii) If L \models Φ is a clause in Π, then NOT(L) \models neg(Φ) (the dual definition) is in Π'.

The extension EXT(Π, neg) preserves the meaning of Π only if Π is compact. Program Π is compact if for every clause L \models Φ in Π there is no other clause L' \models Φ' such that L and L' are unifiable. In the simple case when a program is not compact due to the presence of clauses L \models Φ and L \models Φ', they may be combined to form a clause L \models Φ ∨ Φ'. This reduction does
Constructive Negation

not alter the least model of the program, as demonstrated in lemma 7.2, below. When the heads of the clauses are not identical, the problem of combining and separating the definitions is more difficult, but solvable.

Proof that clauses $L \triangleright \Phi$ and $L \triangleright \Phi'$ are equivalent to clause $L \triangleright \Phi \lor \Phi'$ is quite simple, and is presented next.

Lemma 7.2: Let $\Pi_1 = \Pi \cup \{L \triangleright \Phi, L \triangleright \Phi'\}$, and $\Pi_2 = \Pi \cup \{L \triangleright \Phi \lor \Phi'\}$, where $\Pi$ is a program. Then $M(\Pi_1) = M(\Pi_2)$.

Proof: The proof is in two parts, each utilizing induction on the structure of the least models, demonstrating first inclusion of $M(\Pi_1)$ in $M(\Pi_2)$ and then inclusion of $M(\Pi_2)$ in $M(\Pi_1)$.

$M(\Pi_1) \subseteq M(\Pi_2)$:

(Basis) $true \in M(\Pi_1)$ and $true \in M(\Pi_2)$.

(Induction) Assume $M(\Pi_1) \subseteq M(\Pi_2)$. Let $\sigma$ be a substitution such that $\sigma(L) \triangleright \sigma(\Phi)$ and $\sigma(L) \triangleright \sigma(\Phi')$ are ground instances. If $\Pi_1 \models \sigma(\Phi)$ or $\Pi_1 \models \sigma(\Phi')$, then $\sigma(L) \in M(\Pi_1)$. By the hypothesis, $\Pi_2 \models \sigma(\Phi)$ or $\Pi_2 \models \sigma(\Phi')$, so $\sigma(L) \in M(\Pi_2)$.

$M(\Pi_2) \subseteq M(\Pi_1)$:

(Basis) $true \in M(\Pi_2)$ and $true \in M(\Pi_1)$.

(Induction) Assume $M(\Pi_2) \subseteq M(\Pi_1)$. If $\sigma(L) \triangleright \sigma(\Phi) \lor \sigma(\Phi')$ is a ground clause and $\Pi_2 \models \sigma(\Phi)$ or $\Pi_2 \models \sigma(\Phi')$, then $\sigma(L) \in M(\Pi_2)$. By the hypothesis, $\Pi_1 \models \sigma(\Phi)$ or $\Pi_1 \models \sigma(\Phi')$, so $\sigma(L) \in M(\Pi_1)$. □

I can now demonstrate why programs must be compact. Assume a program $\Pi$ contains two clauses:

(i) $p :- a$.

(ii) $p :- b$.

Clearly, this program is not compact. By lemma 7.2, these clauses have the same effect as the single clause:
Constructive Negation

(i') \( p : - a \lor b. \)

The extension of \( \Pi \) introduces two new clauses:

\[ \neg p : - \neg a. \]
\[ \neg p : - \neg b. \]

Again using lemma 7.2, these new clauses have the same effect as the single clause:

\[ \neg p : - \neg a \lor \neg b. \]

Also the extension of clause (i') introduces the new clause:

\[ \neg p : - \neg a \land \neg b. \]

Clearly the two definitions of \( \neg p \) do not have the same effect. Thus extension of a non-compact program does not preserve its model-theoretic meaning.

I am now prepared to state the main result of this section:

Lemma 7.3: \( \Pi' = \text{EXT}(\Pi, \text{neg}) \) is consistent if:

(i) \( \Pi \) does not contain definitions for \( \text{true} \) and \( \neg \text{true} \);

(ii) The necessary condition for consistency holds for \( \Pi \);

(iii) \( \Pi \) is compact.

Proof: By induction on subsets \( M_i \) of the least model \( M(\Pi') \).

(Basis) Since \( \text{true} \in M_0 \) and \( \neg \text{true} \notin M_0 \), the basis case is consistent.

(Induction) Assume \( M_i \) is consistent for \( i < n \). Inconsistency can be generated within \( M_n \) in the following ways:

(Case 1) There is a ground instance \( \neg \text{true} : - \Phi \) of a clause in \( \Pi \) and \( M_i \models \Phi \). But \( \neg \text{true} : - \Phi \) cannot be an instance of a clause in \( \Pi \), since this is prohibited by restriction (i). Also \( \text{EXT}(\Pi, \text{neg}) \) does not generate this clause, since \( \Pi \) would then contain a clause of the form \( \text{true} : - \Phi' \), which is also prohibited by restriction (i).
Constructive Negation

(Case 2) There are ground instances \( L \vdash \Phi \) and \( \text{NOT}(L) \vdash \Phi' \) of clauses in \( \Pi' \). By the necessary condition for consistency, these clauses cannot both be instances of clauses in \( \Pi \). So assume only \( L \vdash \Phi \) is an instance of a clause in \( \Pi \). Because \( \Pi \) is compact, there is no other ground instance of a clause in \( \Pi \) of the form \( L \vdash \Phi'' \), where \( \Phi'' \neq \Phi \). Hence, it must be that \( \Phi' = \text{neg}(\Phi) \). By the definition of the negation function \( \text{neg} \), when \( M_i \) is consistent, \( M_i \models \Phi \) implies \( M_i \not\models \Phi' \). By hypothesis, \( M_i \) is consistent, so a contradiction occurs: \( L \in M_n \) while \( \text{NOT}(L) \not\in M_n \). □

Hence, when a program \( \Pi \) is compact and satisfies the necessary condition for consistency, \( \text{EXT}(\Pi, \text{neg}) \) forms a conservative extension of \( \Pi \).

To illustrate the conservative extension of a program, consider the program below:

**Example 7.4**

(i) \( \text{subsequence}(\text{nil}, M) : - \text{true} \).  
(ii) \( \text{subsequence}(\text{cons}(X, L), \text{cons}(Y, M)) : - \text{eq}(X, Y) \land \text{subsequence}(L, M) \).  
(iii) \( \text{subsequence}(\text{cons}(X, L), \text{cons}(Y, M)) : - \text{subsequence}(\text{cons}(X, L), M) \).  
(iv) \( \neg\text{subsequence}(\text{cons}(X, L), \text{nil}) : - \text{true} \).

This program defines the subsequence predicate as described for example 4.1. The program in example 7.4 satisfies the necessary condition for consistency. However, the program is not compact, due to the presence of clauses (ii) and (iii). These clauses can be combined to form the following clause:

\[
\text{subsequence}(\text{cons}(X, L), \text{cons}(Y, M)) : - \\
[\text{eq}(X, Y) \land \text{subsequence}(L, M)] \\
\lor \text{subsequence}(\text{cons}(X, L), M) .
\]

Finally, using the rules characterizing the NEG relation presented above, the program presented in example 4.1 results.
Constructive Negation

8. Summary

This paper serves as an introduction to a logic employing constructive negation. I have demonstrated the soundness and completeness of this logic. I have also argued that constructive negation fits within the procedural semantics of resolution better than negation by failure. Finally, I described the conservative extension of a program. Use of a conservative extension not only prevents formulation of inconsistent programs, it also produces negative definitions from positive definitions. Elaboration of the properties of the conservative extension EXT(Π, neg) remains to be done. These properties will rely on a detailed description of the procedural semantics of universal quantification.
Appendix A. Equivalence of SLDCN and SLD Procedures

The syntax chosen for extended Horn-clause programs is slightly expanded from the standard Horn-clause syntax. Standard Horn-clause syntax does not include the following elements:

- negative literals,
- disjunctive phrases,
- existentially quantified phrases.

In all other respects the syntactic forms are identical. Likewise, the SLD procedure is identical to the SLDCN procedure, absent rules for the elements listed above. Extended Horn-clauses will be convenient for describing certain properties of constructive negation. However, it is important to know if a standard Horn-clause interpreter, using SLD-resolution, can successfully execute an extended Horn-clause logic program. This proof is conducted in this section.

A first step in executing extended logic programs with a standard interpreter is to translate extended logic programs into standard logic. This is achieved using a syntactic transformation function \( \text{TRANS} \), mapping phrases to phrases. If \( A \) is an atom, and \( \Phi_1 \) and \( \Phi_2 \) are phrases, the recursive definition of \( \text{TRANS} \) is as follows:

(i) \( \text{TRANS}[A] = A \).
(ii) \( \text{TRANS}[\neg A] = \neg \text{TRANS}[A] \).
(iii) \( \text{TRANS}[\Phi_1 \land \Phi_2] = \text{TRANS}[\Phi_1] \land \text{TRANS}[\Phi_2] \).
(iv) \( \text{TRANS}[\Phi_1 \lor \Phi_2] = \text{TRANS}[\Phi_1] \lor \text{TRANS}[\Phi_2] \).
(v) \( \text{TRANS}[(\exists x)\Phi] = \exists x(\text{TRANS}[\Phi]) \).

\( \text{TRANS} \) can be extended naturally to map over clauses, and finally over an entire program. Substitution commutes with translation: \( \text{TRANS}[\sigma(\Phi)] = \sigma(\text{TRANS}[\Phi]) \).

Execution of translated standard Horn-clause programs by the SLD procedure is achieved by introducing the following clauses, together designated later as \( \Gamma \):

- 22 -
Constructive Negation

\( \exists \text{exists}(F) \Rightarrow F. \)

\( \text{or}(F_1,F_2) \Rightarrow F_1. \)

\( \text{or}(F_1,F_2) \Rightarrow F_2. \)

These rules will be used to perform certain higher-order deductions by the SLD-resolution procedure.

Proof of equivalence between SLD and SLDCN procedures relies on their completeness properties. Completeness of SLDCN was demonstrated in the previous section. Completeness of SLD has been shown in [EK76]. Let \( \Pi \) be an extended Horn-clause program with the following restrictions:

(i) \( \Pi \) contains no clauses defining the predicates \( \text{not} \), \( \text{exists} \), or \( \text{or} \).

(ii) Every existentially quantified variable has a distinct name.

Then let \( \Pi' = \text{TRANS}[\Pi] \cup \Gamma \). The following theorem demonstrates equivalence of least models generated by extended and standard Horn-clause logic programs.

Lemma A.1: \( \Pi \models \Phi \) if and only if \( \Pi' \models \text{TRANS}[\sigma(\Phi)] \), for all ground phrases \( \Phi \), and some substitution \( \sigma \).

Proof: To show that \( \Pi \models \Phi \) implies \( \Pi' \models \text{TRANS}[\sigma(\Phi)] \), proceed by induction on the structure of the least model \( M(\Pi) \).

(Basis) \( \text{true} \in M(\Pi) \), and \( \text{true} \in M(\Pi') \).

(Induction) Assume the lemma is true for ground phrases \( \Phi_1 \) and \( \Phi_2 \) such that \( \Pi \models \Phi_1 \) and \( \Pi \models \Phi_2 \). Since the translation function \( \text{TRANS} \) has no effect for atoms and conjunctions, these cases are trivial, and will not be considered.

\( \Phi = \neg A \) Then \( \text{TRANS}[\neg A] = \text{not}(A) \). \( \Pi \models \neg A \) when there is a clause \( \neg A \Rightarrow \Phi \) in \( \Pi \) such that \( \rho(A) = A \), \( \rho(\Phi) = \Phi_1 \), and \( \Pi \models \Phi_1 \). The clause \( \text{not}(A) \Rightarrow \text{TRANS}(\Phi) \) is contained in \( \Pi' \). By the hypothesis, \( \Pi' \models \text{TRANS}[\Phi_1] \), so \( \Pi' \models \text{not}(A) \).

\( \Phi = \Phi_1 \lor \Phi_2 \) Then \( \text{TRANS}[\Phi_1 \lor \Phi_2] = \text{or}(\text{TRANS}[\Phi_1], \text{TRANS}[\Phi_2]) \). \( \Pi \models \Phi_X \) for \( X \in \{1, 2\} \). If
Constructive Negation

$X = 1$, there is a clause $or(F_1,F_2) :- F_1$ in $\Gamma$, and an instance of this clause is $or(TRANS[\Phi_1], TRANS[\Phi_2]) :- TRANS[\Phi_1]$. By the hypothesis, $\Pi \models TRANS[\Phi_1]$, so $\Pi \models or(TRANS[\Phi_1], TRANS[\Phi_2])$. Similarly for $X = 2$.

$[\Phi = (3z)\Phi_1]$ Then $TRANS((3z)\Phi_1) = exists(TRANS[\Phi_1])$. $\Pi \models \Phi$, so $\Pi \models \sigma(\Phi_1)$ for some $\sigma = [t/z]$. There is a clause $exists(F) :- F$ in $\Gamma$, and an instance of this clause is $exists(TRANS[\sigma(\Phi_1)]) :- TRANS[\sigma(\Phi_1)]$. By the hypothesis, $\Pi \models TRANS[\sigma(\Phi_1)]$, so $\Pi \models exists(TRANS[\sigma(\Phi_1)])$.

Proof of the converse is achieved in the same manner. □

Theorem A.2: When $\Pi$ is a program observing the restrictions presented above, $\Phi$ is deducible from $\Pi$ by SLDCN if and only if it is also deducible from program $TRANS[\Pi] \cup \Gamma$ by SLD.

Proof: As a consequence of the completeness theorems for SLD and SLDCN, and use of lemma A.1.
Constructive Negation

References


