EQUIVALENT LOGIC PROGRAMS

CHAN KWOK HUNG

The development of logic programs sometimes takes the following course. We begin with an intuitively correct program and then replace it by an equivalent program with better performance. This article introduces the notion of CAS-equivalent logic programs: logic programs with identical correct answer substitutions with respect to a set of predicate symbols. A least model criterion for CAS equivalence is suggested, and the correctness of the criterion is proved. The least model approach is illustrated by the CAS equivalence of the definition of the reverse relation by tail recursion and the intuitive definition of reverse in terms of append. Based on the fixed point semantics of CAS-equivalent logic programs, we may provide formal proofs of the correctness of PROLOG meta interpreters. A PROLOG meta interpreter E-Chain for logic programs with equality is presented, and its correctness is established.

1. INTRODUCTION

A virtue of logic programming is that the language of specification may be used as the language of implementation [10, 8]. Hence we may use theories in first order languages that lucidly formulate our intuitive understanding of the problem domains as programs. However, the use of such programs frequently leads to logic programming systems with unacceptable performance. A general approach to this problem is that we begin with a logic program \( \mathcal{P} \) that gives a lucid description of the problem domain in question and then replace \( \mathcal{P} \) by a program \( \mathcal{P}' \) that is equivalent to \( \mathcal{P} \) but with better performance. (See [15, Chapter VI] for an exposition of formal program synthesis and references.)
Equivalent Logic Programs

Consider the familiar example of the reverse relation. The intended interpretation of \( \text{reverse}(L_1, L_2) \) is that \( L_2 \) is the reversal of the list \( L_1 \). The following program \( \text{REV} \) gives an intuitive definition of reverse in terms of append:

\[
\begin{align*}
\text{REV} & : \\
\text{reverse}([], []). & \leftarrow (1) \\
\text{reverse}([H|T], L) & \leftarrow \text{reverse}(T, RT), \\
& \quad \text{append}(RT, [H], L). & \leftarrow (2) \\
\text{append}([], I, I). & \leftarrow (3) \\
\text{append}([H|T], L, [H|L_1]) & \leftarrow \text{append}(T, L, L_1). & \leftarrow (4)
\end{align*}
\]

However, the construction of reversals using append is an unnecessarily inefficient process: \((n^2 + 3n + 2)/2\) resolution steps are required for a list of \( n \) elements. The number of resolution steps required can be reduced to \( n + 2 \) if we define reverse by tail recursion as follows:

\[
\begin{align*}
\text{REV1} & : \\
\text{reverse}(L, RL) & \leftarrow \text{reverse1}(L, [], RL). & \leftarrow (5) \\
\text{reverse1}([], L, L). & \leftarrow (6) \\
\text{reverse1}([H|T], L, NL) & \leftarrow \text{reverse1}(T, [H|L], NL). & \leftarrow (7)
\end{align*}
\]

The intended interpretation of \( \text{reverse1}(L_1, L_2, L_3) \) is: \( L_3 \) is obtained by appending \( L_2 \) to the reversal of \( L_1 \). If \( L_2 \) is the empty list, then \( L_3 \) is precisely the reversal of \( L_1 \). The computation of the reversal of \( L \) is straightforward using \( \text{REV1} \).

Although \( \text{REV1} \) and \( \text{REV} \) are obviously not logically equivalent, they are equivalent in the following sense: for all subgoals \( G \) of the form \( \text{reverse}(l, rl) \), the set of correct answer substitutions for \( \text{REV1} \cup \{ G \} \) is identical to the set of correct answer substitutions for \( \text{REV} \cup \{ G \} \). We say that \( \text{REV1} \) and \( \text{REV} \) are correct answer substitution equivalent (CAS-equivalent) with respect to the predicate \( \text{reverse} \). In this article we will develop the fixed point semantics of CAS equivalence for definite clause logic programs. (See [21,22] for some other notions of equivalent logic programs.) A least model criterion for two logic programs to be CAS-equivalent with respect to a set of predicates is suggested. Based on this criterion it is not difficult to construct formal proofs of CAS equivalence between logic programs.

Prolog Metainterpreters

Based on the fixed point semantics of CAS-equivalent logic programs, we may provide formal proofs of the correctness of PROLOG metainterpreters. Let us consider the case of symmetry as an example. Although the symmetry axiom

\[
\text{sym: } \text{R}(x, y) \leftarrow \text{R}(y, x)
\]

formulates our intuitive understanding of symmetry, the inclusion of \( \text{sym} \) in a logic program introduces infinite loops in PROLOG. A well-known solution to this kind of problem is to replace the axioms yielding infinite loops by inference rules. For example, we can extend SLD resolution by the following rule:

\[
\text{SYM: } \text{Let} \\
G & \leftarrow \text{R}(s, t), A_2, \ldots, A_m
\]
be a goal, and let
\[ C: ~ R(u, v) \leftarrow B_1, \ldots, B_q \]
be an input clause sharing no variables with \( G \). Then the new goal
\[ G': \leftarrow (B_1, \ldots, B_q, A_2, \ldots, A_m)\theta \]
is derived from \( G \) and \( C \) using the unifier \( \theta \) via the leftmost selection rule if \( \theta \) is an mgu of the pair \( \langle R(s, t), R(u, v) \rangle \) or of the pair \( \langle R(s, t), R(v, u) \rangle \).

An extension of SLD-resolution by the rule \( SYM \) is called symmetric SLD resolution. Symmetric SLD resolution can be implemented by the PROLOG metainterpreter \( SYMSLD \) introduced below.

A PROLOG metainterpreter is a PROLOG program that simulates the actions of an inference machine. (The term “metainterpreter” has also been used in a more restrictive way referring to programs that treat other programs as data, programs that transform and simulate other programs [25, 2].) In the case of symmetric SLD resolution, resolution of subgoals involving the predicate \( R \) is modified according to the rule \( SYM \). We can simulate the effect of \( SYM \) without modifying the interpreter of PROLOG as follows: (1) pick a new predicate \( R' \), and transform a program \( \mathcal{P} \) for symmetric SLD resolution into a program \( \mathcal{P}' \) which is identical to \( \mathcal{P} \) except that every occurrence of \( R \) in the heads of program clauses is replaced by an occurrence of \( R' \); (2) we supplement \( \mathcal{P}' \) by the following program:
\[ SYMSLD: \begin{align*}
R(x, y) & \leftarrow R'(x, y) \\
R(x, y) & \leftarrow R'(y, x)
\end{align*} \]
The program \( SYMSLD \) in effect acts as a metainterpreter for \( \mathcal{P} \). The procedural interpretation of PROLOG programs provides an intuitive justification of the fact that \( SYMSLD \) implements the inference rule \( SYM \). Given a procedure call \( R(s, t) \), the metainterpreter first calls \( R'(s, t) \), then it switches the position of \( s \) and \( t \) and calls \( R'(t, s) \). Infinite loops caused by the symmetry axiom are avoided.

\( \mathcal{P}' \cup SYMSLD \) and \( \mathcal{P} \cup sym \) are also equivalent from the user's point of view. Since \( R' \) and \( SYMSLD \) are hidden from the user, \( R' \) does not occur in a user-supplied query. (We may treat \( R' \) as a reserved symbol and forbid the use of \( R' \) in user-supplied programs and queries.) Accordingly, the extension of \( R' \) makes no difference to the user as long as the extensions of other predicates are correct.

In Section 4 we extend \( SYMSLD \) to a metainterpreter \( EChain \) for logic programs with equality. A proof of CAS equivalence can be used to establish that \( EChain \) subsumes the symmetry, transitivity, and predicate substitutivity axioms of equality.

2. CAS-EQUIVALENT LOGIC PROGRAMS

We develop in this section the fixed point semantics of CAS equivalent logic programs. (See [11, 1, 19] for the foundations of the fixed point semantics of logic programs, and [22, 6] for other notions of equivalent logic programs.) The notion of correct answer substitution is formally defined as follows.
Definition 2.1 (Correct answer substitution [19, §4]). Let \( \mathcal{P} \) be a program in \( L_{\mathcal{P}} \), and let

\[ G: \leftarrow A_1, \ldots, A_m \]

be a goal. An answer substitution \( \theta \) in \( L_{\mathcal{P}} \) for \( \mathcal{P} \cup \{ G \} \) is a substitution for the variables in \( G \). \( \theta \) is called a correct answer substitution iff

\[ \mathcal{P} = \forall \left( (A_1 \land \cdots \land A_m) \theta \right) \]

Based on the notion of correct answer substitution, we introduce the notion of CAS equivalence as follows:

Notation. Let \( \Xi \) be a set of predicate symbols. An atom \( P(t_1, \ldots, t_n) \) is called a \( \Xi \)-atom iff \( P \in \Xi \). A goal

\[ G: \leftarrow A_1, \ldots, A_m \]

is called a \( \Xi \)-goal iff each atom \( A_i \) in \( G \) is a \( \Xi \)-atom.

Definition 2.2 (CAS equivalence with respect to \( \Xi \)). Let \( \mathcal{P} \) and \( \mathcal{P}' \) be programs. Let \( \Xi \) be a set of predicate symbols. \( \mathcal{P} \) and \( \mathcal{P}' \) are said to be CAS-equivalent with respect to \( \Xi \) iff for all \( \Xi \)-goals \( G \) the set of correct answer substitutions in \( L_{\mathcal{P}} \) for \( \mathcal{P} \cup \{ G \} \) is identical to the set of correct answer substitutions in \( L_{\mathcal{P}'} \) for \( \mathcal{P}' \cup \{ G \} \).

(This relative notion of CAS equivalence is an extension of the corresponding notion in [3, 6].)

In [3, 6] it is shown that if two programs \( \mathcal{P} \) and \( \mathcal{P}' \) have the same least model, then \( \mathcal{P} \) and \( \mathcal{P}' \) are refutational equivalent in the following sense: for all goals \( G \), \( \mathcal{P} \cup \{ G \} \) is unsatisfiable iff \( \mathcal{P}' \cup \{ G \} \) is unsatisfiable. However, logic programs with the same least model may not have the same set of correct answer substitutions. Consider the following programs in a language \( L \) with \( a \) as the only constant symbol.

Example 2.1.

\[ \mathcal{P}: \]
\[ P(a). \]
\[ Q(a). \]

\[ \mathcal{P}': \]
\[ P(x). \]
\[ Q(a). \]

\( \mathcal{P} \) and \( \mathcal{P}' \) have the same least model: \( \{ P(a), Q(a) \} \). However, \( \emptyset \) is not a correct answer substitution for \( \mathcal{P} \cup \{ \leftarrow P(x) \} \), although it is a correct answer substitution for \( \mathcal{P}' \cup \{ \leftarrow P(x) \} \). The crucial point is that we cannot infer \( \forall P(x) \) from the fact that every ground instance of \( P(x) \) is in the least model of the program.

---

\(^1\forall(C)\) is the universal closure of \( C \).
In studying the semantics of logic programs, it is a common practice to focus attention on the language $L_{\varphi}$ of the program $\varphi$ under consideration: the sets of constant, function, and predicate symbols in $L_{\varphi}$ are precisely the sets of constant, function, and predicate symbols in $\varphi$ (except in the case where $\varphi$ contains no constants). By relaxing this restriction and considering finite extensions of $L_{\varphi}$ we can develop a least model criterion for CAS equivalence (Proposition 2.2). The notion of a finite extension is defined as follows.

**Definition 2.3 (Finite extension).** Let $\varphi$ be a program. A finite extension of $L_{\varphi}$ is a language $L$ identical to $L_{\varphi}$ except that the sets of constant, function, and predicate symbols in $L$ are $\mathcal{C}_{\varphi} \cup \mathcal{C}$, $\mathcal{F}_{\varphi} \cup \mathcal{F}$, and $\mathcal{P}_{\varphi} \cup \mathcal{P}$, where $\mathcal{C}_{\varphi}$, $\mathcal{F}_{\varphi}$, and $\mathcal{P}_{\varphi}$ are the sets of constant, function, and predicate symbols in $\varphi$, and where $\mathcal{C}$, $\mathcal{F}$, and $\mathcal{P}$ are finite sets of new constant, function, and predicate symbols not occurring in $\varphi$.

**Notation ([A]).** Let $A$ be an atom in a language $L$. Then $[A]$ denotes the set of all ground instances of $A$ in $L$.

**Notation ($M_L(\varphi)$).** Let $\varphi$ be a program in $L$. Then $M_L(\varphi)$ denotes the least model of $\varphi$ in $L$.

We have the following proposition, which provides a criterion for universal generalization.

**Proposition 2.1 (Generalization).** Let $\varphi$ be a program. Let $A_1, \ldots, A_m$ be atoms in $L_{\varphi}$. Let $x_1, \ldots, x_n$ be the variables in $A_1, \ldots, A_m$. If there is a finite extension $L$ of $L_{\varphi}$ with at least $n$ new constants and $[A_1], \ldots, [A_m] \subseteq M_L(\varphi)$, then $\varphi = \forall(A_1 \wedge \cdots \wedge A_m)$.

(Cf. the Theorem on Constants in [24].)

**Proof.** (By reductio ad absurdum.) Let $L$ be a finite extension of $L_{\varphi}$ with at least $n$ new constants. Suppose $[A_1], \ldots, [A_m] \subseteq M_L(\varphi)$. Assume that $\varphi \neq \forall(A_1 \wedge \cdots \wedge A_m)$. Then $\varphi \cup \{\neg \forall(A_1 \wedge \cdots \wedge A_m)\}$ is satisfiable, $\neg \forall(A_1 \wedge \cdots \wedge A_m)$ is logically equivalent to $\exists(-A_1 \lor \cdots \lor -A_m)$.

Let $\{c_1, \ldots, c_n, \ldots, c_k\}$ be the set of new constant symbols in $L$. Let $\theta$ be the substitution $\{c_1/x_1, \ldots, c_n/x_n\}$. Then $\varphi \cup \{\neg A_1 \lor \cdots \lor \neg A_m\}\theta$ is satisfiable. (See, e.g., [20, §1.5] or any standard logic text on Skolem functions.) Accordingly, there is a Herbrand model $I_L$ of $\varphi$ in which $(\neg A_1 \lor \cdots \lor \neg A_m)\theta$ is true. It follows that there is a $j, 1 \leq j \leq m$, such that $A_j\theta \in I_L \supseteq M_L(\varphi)$. Hence $A_j\theta \in M_L(\varphi)$, contradicting the assumption that $[A_j] \subseteq M_L(\varphi)$. □

Based on Proposition 2.1, we can prove the following proposition which provides a least model criterion for CAS equivalence.

---

$^2 \exists(C)$ is the existential closure of $C$.  

Notation. Let $\mathcal{P}$ be a program in $L$. Let $\Xi$ be a set of predicate symbols. Then

$$\Xi_L(\mathcal{P}) = \{ P(t_1, \ldots, t_n) \in M_L(\mathcal{P}) | P \in \Xi \}.$$  

$\Xi_L(\mathcal{P})$ is called the $\Xi$-restriction of the least model of $\mathcal{P}$ in $L$.

Proposition 2.2 (CAS-equivalence). Let $\mathcal{P}$ and $\mathcal{P}'$ be programs. Let $\Xi$ be a set of predicate symbols. $\mathcal{P}$ and $\mathcal{P}'$ are CAS-equivalent with respect to $\Xi$ if for every nonnegative integer $k$ there is a finite extension $L$ of $L_{\mathcal{P} \cup \mathcal{P}'}$ with $k$ new constants such that

$$\Xi_L(\mathcal{P}) = \Xi_L(\mathcal{P'}).$$

Proof: Let

$$G: \leftarrow A_1, \ldots, A_m$$

be an arbitrary $\Xi$-goal. Let $x_1, \ldots, x_n$ be the variables in $G$. Let $L$ be a finite extension of $L_{\mathcal{P} \cup \mathcal{P}'}$ with $n$ new constant symbols such that $\Xi_L(\mathcal{P}) = \Xi_L(\mathcal{P}')$. Let $\theta$ be a correct answer substitution in $L_{\mathcal{P}}$ for $\mathcal{P} \cup \{G\}$. By Definition 2.1

$$\mathcal{P} \models \forall((A_1 \land \cdots \land A_m) \theta).$$

Since the least model of $\mathcal{P}$ is identical to the set of ground atoms implied by $\mathcal{P}$[11], we have

$$[A_1 \theta], \ldots, [A_m \theta] \subseteq \Xi_L(\mathcal{P}) = \Xi_L(\mathcal{P}') \subseteq M_L(\mathcal{P}').$$

By Proposition 2.1,

$$\mathcal{P}' \models \forall(A_1 \land \cdots \land A_m \theta)$$

$$\vdash \forall((A_1 \land \cdots \land A_m) \theta).$$

Hence $\theta$ is a correct answer substitution for $\mathcal{P}' \cup \{G\}$.

By the same argument, if $\theta$ is a correct answer substitution in $L_{\mathcal{P}'}$ for $\mathcal{P}' \cup \{G\}$, then $\theta$ is a correct answer substitution in $L_{\mathcal{P}}$ for $\mathcal{P} \cup \{G\}$.

3. EQUIVALENT DEFINITIONS OF reverse

We outline in this section a formal proof of CAS equivalence between the programs $REVI$ and $REV$ with respect to the reverse relation (Section 1). The proof is based on the following results concerning the append relation defined in the program $REV$ and the relationship between reverse1 (as defined in $REVI$) and reverse (as defined in $REV$). These results can be established by inductive proofs.

Let $L$ be an arbitrary finite extension of $L_{REV}$. For all lists $l_1$, $[h|l_1]$, and $l_2$, $append(l_1,[h|l_1],l_2) \in M_L(REV)$ iff there is a list $nl$ such that $append((l_1,[h],nl)$. $append(nl,t,tl) \in M_L(REV)$.

Let $L$ be an arbitrary finite extension of $L_{REV}$. For all lists $l_1$ and $l_2$, $append([1,[],l_2) \in M_L(REV)$ iff $l_1$ and $l_2$ are syntactically identical.
Let \( L \) be an arbitrary finite extension of \( L_{REV \cup REVI} \). For all lists \( l, l1, \) and \( nl \), \( reverse(l, l1, nl) \in M_E(REVI) \) iff there is a list \( rl \) such that \( reverse(l, rl) \) \( append(rl, l1, nl) \in M_E(REV) \).

We have the following proposition.

Proposition 3.1. The programs \( REV \) and \( REVI \) are CAS-equivalent with respect to the set \{ reverse \}.

Proof. Let \( \Xi \) be \{ reverse \}, and let \( L \) be an arbitrary finite extension of \( L_{REV \cup REVI} \). Based on the preceding results, we can establish that \( \Xi_L(REV) \) and \( \Xi_L(REVI) \) are identical. The CAS-equivalence of \( REV \) and \( REVI \) then follows from Proposition 2.2. \( \square \)

4. A META-INTERPRETER FOR LOGIC PROGRAMS WITH EQUALITY

In this section we extend symmetric SLD resolution to a metainterpreter for logic programs with equality. The inference machine of the metainterpreter is called E-Chain. We can establish the correctness of E-Chain by a proof of CAS equivalence.

E-Chain

The logical characteristics of equality are captured by the following axioms:

- ref: \( x = x \)
- sym: \( x = y \leftrightarrow y = x \)
- tran: \( x = y \leftrightarrow x = z, z = y \)
- Pred: \( \{ P(x_1, \ldots, x_n) \leftrightarrow P(y_1, \ldots, y_n), \quad x_1 = y_1, \ldots, x_n = y_n \mid P \text{ is an } n\text{-place predicate symbol in } L \} \)
- Fun: \( \{ f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \leftrightarrow x_1 = y_1, \ldots, x_n = y_n \mid f \text{ is an } n\text{-place function symbol in } L \} \)

ref, sym, tran, Pred, and Fun are the reflexivity, symmetry, transitivity, predicate substitutivity, and function substitutivity axioms, respectively [20]. We use \( \sigma \) to denote the set \{ ref, sym, tran \} \( \cup \) Pred \( \cup \) Fun.

Including sym, tran, and Pred in a logic program causes infinite loops, and the aim of extending SLD resolution to the more powerful system E-Chain is to subsume the symmetry, transitivity, and predicate substitutivity axioms by rules of inference.

A derivation step in E-Chain is characterized as follows: Let \( G \) be a goal of the form

\[ \leftarrow P(s_1, \ldots, s_n), A_2, \ldots, A_m \]

and \( P(s_1, \ldots, s_n) \) be the subgoal selected by the leftmost selection rule.
1. \( P \) is \( '=' \). In this case the selected subgoal is an equation \( s_1 = s_2 \). The equation \( s_1 = s_2 \) is then established by constructing an equality chain from \( s_1 \) to \( s_2 \). (The notion of an equality chain is explained below.) Let \( \theta \) be the substitution of the equality chain constructed. The new goal

\[
\leftarrow (A_2, \ldots, A_m) \theta
\]

is derived from \( G \) using the unifier \( \theta \).

2. \( P \) is not \( '=' \). Let

\[
C: \quad P(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_q
\]

be an input clause from the program. Then the new goal

\[
\leftarrow s_1 = t_1, \ldots, s_n = t_n, B_1, \ldots, B_q, A_2, \ldots, A_m
\]

is derived from \( G \) and \( C \).

Part 2 of the rule subsumes the predicate substitutivity axioms, while part 1 subsumes the reflexivity, symmetry, and transitivity axioms of equality.

Let us explain the idea of an equality chain (abbreviated as e-chain) in detail. Since equality is reflexive, symmetric, and transitive, it is an equivalence relation. We have the following result, which holds for all equivalence relations:

**Notation** (\( \equiv \)). Let \( E_1 \) and \( E_2 \) be expressions in \( L \). Then \( E_1 \equiv E_2 \) iff \( E_1 \) is syntactically identical to \( E_2 \).

**Proposition 4.1.** A ground equation \( s = t \) is a logical consequence of \( \mathcal{P} \cup \{ \text{ref, sym, tran} \} \) iff there is a finite sequence of ground terms

\[
r_0, r_1, \ldots, r_l
\]

satisfying the following conditions:

1. \( s \) is \( r_0 \), \( t \) is \( r_l \);
2. for each \( i, 1 \leq i \leq l \), there is a clause

\[
C_i: \quad u = v \leftarrow A_1, \ldots, A_m
\]

in \( \mathcal{P} \) and a substitution \( \theta \) such that \( A_1 \theta, \ldots, A_m \theta \) are ground atoms logically implied by \( \mathcal{P} \cup \{ \text{ref, sym, tran} \} \) and either

a. \( r_{i-1} \equiv u \theta \) and \( r_i \equiv v \theta \), or
b. \( r_{i-1} \equiv v \theta \) and \( r_i \equiv u \theta \).

This result may be proved by induction on \( T_{\mathcal{P} \cup \{ \text{ref, sym, tran} \} \uparrow} \). The essential idea is that there is a chain of ground equations linking \( s \) and \( t \) where each link of the chain is justified by an equality clause: a program clause with an equation in its head. Let \( \mathcal{P} \) be a user-supplied program. A pair of ground terms \( \langle s, t \rangle \) is called an e-link iff there is an equality clause

\[
u = v \leftarrow A_1, \ldots, A_m\]
in $\mathcal{P}$ and a substitution $\theta$ such that

$\mathcal{P} \cup \{ \text{ref, sym, tran} \} \models A_1\theta, \ldots, A_m\theta$

and either $s \equiv u\theta$, $t \equiv v\theta$ or $s \equiv u\theta$, $t \equiv u\theta$. And a sequence

$r_0, r_1, \ldots, r_l$

of ground terms is called an e-chain for the pairs of terms $\langle r_i, r_j \rangle$ iff $l \geq 0$ and for each $i$, $1 \leq i \leq l$, $\langle r_{i-1}, r_i \rangle$ is an e-link.

Part 2 of the rule may be implemented by transforming nonequality clauses in a program $\mathcal{P}$ to their homogeneous form.

**Definition 4.1.** Let

$$C: \quad Q(t_1, \ldots, t_n) \leftarrow A_1, \ldots, A_m$$

be a clause. The **homogeneous form** of $C$ is

$$h(C): \quad Q(x_1, \ldots, x_n) \leftarrow x_1 = t_1, \ldots, x_n = t_n, A_1, \ldots, A_m$$

where $x_1, \ldots, x_n$ are distinct variables not occurring in $C$.

Part 1 of the rule may be implemented as follows. First we pick a new predicate $\neq *$ and replace every occurrence of the equality predicate $=$ in the head of an equality clause by $\neq *$. Then we supplement the resulting program by the following program:

**EChain:**

$$x = y \leftarrow echain(x, y).$$

$$echain(x, x).$$

$$echain(x, y) \leftarrow elink(x, z), echain(z, y).$$

$$elink(x, y) \leftarrow x = *y.$$

$$elink(x, y) \leftarrow y = *x.$$

where $echain$ and $elink$ are distinct predicate symbols new to $\mathcal{P}$.

To sum up, a user-supplied program $\mathcal{P}$ is transformed into a program

$$\mathcal{P}^* = \{ h(C) \mid C \text{ is a nonequality clause in } \mathcal{P} \}$$

$$\cup \{ s = *t \leftarrow B \mid s = t \leftarrow B \in \mathcal{P} \}$$

$$\cup \text{EChain}. $$

$\mathcal{P}^*$ is then interpreted by the standard PROLOG interpreter. (In this article we have not given a formal definition of $E$-Chain and a formal proof that the program transformation $*$ implements $E$-Chain in SLD resolution. Interested reader may refer to Chan [3,5], where semantic reduction is introduced as a general approach for reducing the semantics of extensions of SLD resolution to that of SLD resolution. The central idea of semantic reduction is also briefly explained in Section 5 of this article.)

---

3 The homogeneous form transformation is introduced in [12]. The homogeneous forms of logic programs are also used in similar contexts in [9,14,13,7,6].
Let us illustrates, by a few simple examples, how E-Chain works. Consider the following program:

\[ \mathcal{P}: \begin{align*}
P(c). \\
a &= b. \\
c &= b.
\end{align*} \]

\(\mathcal{P}\) is first transformed into the following program:

\[ \mathcal{P}^*: \begin{align*}
P(x) &\leftarrow x = c. \\
a &= \*b. \\
c &= \*b. \\
x &= y &\leftarrow \text{echain}(x, y). \\
\text{echain}(x, x). \\
\text{echain}(x, y) &\leftarrow \text{elink}(x, z), \text{echain}(z, y). \\
\text{elink}(x, y) &\leftarrow x = \*y. \\
\text{elink}(x, y) &\leftarrow y = \*x.
\end{align*} \]

The reflexivity, symmetry, and transivity of = are subsumed by the construction of equality chains:

\[ S1: \begin{align*}
\leftarrow a &= a \\
&\leftarrow \text{echain}(a, a)
\end{align*} \]

\[ S2: \begin{align*}
\leftarrow a &= c \\
&\leftarrow \text{echain}(a, c) \\
&\leftarrow \text{elink}(a, z), \text{echain}(z, c) \\
&\leftarrow a = \*z, \text{echain}(z, c) \\
&\leftarrow \text{echain}(b, c) \\
&\leftarrow \text{elink}(b, z'), \text{echain}(z', c) \\
&\leftarrow z' = \*b, \text{echain}(z', c) \\
&\leftarrow \text{echain}(c, c)
\end{align*} \]

Predicate substitutivity is subsumed by transforming nonequality clauses into their homogeneous forms:

\[ S3: \begin{align*}
\leftarrow P(a) \\
&\leftarrow a = c \\
\ldots
\end{align*} \]

Notice that there is no restriction on \(\mathcal{P}\). \(\mathcal{P}\) may be any logic program in a first order language with equality. In particular, it is not required that the equality theory be separable. The equality theory in \(\mathcal{P}\) is said to be separable if \(\mathcal{P}\) can be partitioned into two parts: a definite clause logic program \(D\) and a definite clause equality theory \(E\) where \(D\) contains no equations and all atoms in \(E\) are equations. This is contrary to the frameworks presented in [23, 12, 17, 18, 16], where the equality theory in a program is required to be separable. (See Chan [3, 5] for details.)

In the next subsection we will establish the correctness of E-Chain by showing that \(\mathcal{P}^*\) is CAS-equivalent to \(\mathcal{P} \cup \{ \text{ref, sym, tran} \} \cup \text{Pred} \) with respect to the set of predicate symbols in \(\mathcal{P}\).
The Correctness of E-Chain

The correctness of E-Chain is established as follows. First, we will define the notion of e-link and e-chain with respect to the transformed program $P^*$. Then we present a proposition relating e-chains and equalities and show that $\equiv$ is an equivalence relation with the predicate substitutivity property. (The proofs of the propositions in this section are rather straightforward and have been omitted. Interested readers should refer to the full version of the paper [4].)

Let $P$ be a program, and let $L$ be an arbitrary finite extension of $L_{P^*}$ ($= L_{P \cup P^*}$).

**Definition 4.2 (e-link).** A pair of ground terms $(s, t)$ is called an e-link iff there is a clause \[ u = ^*v \leftarrow A_1, \ldots, A_m \] in $P^*$ and a substitution $\theta$ such that $A_1\theta, \ldots, A_m\theta \in M(P^*)$ and either

1. $s \equiv u\theta$ and $t \equiv v\theta$, or
2. $s \equiv v\theta$ and $t \equiv u\theta$.

**Definition 4.3 (e-chain).** A sequence \[ S: \quad r_0, r_1, \ldots, r_l \] of ground terms is called an e-chain for the pair of terms $(s, t)$ iff $(r_0, r_1)$ is an e-link and for each $i, 1 \leq i \leq l$, $(r_i, r_{i+1})$ is an e-link. We say that the length of $S$ is $l$. (Note that any ground term standing alone is an e-chain of length 0.)

The following results regarding properties of e-links and e-chains follow directly from Definitions 4.2 and 4.3 and do not depend on any property of $P^*$.

Let $s, t$ be ground terms. $(s, t)$ is an e-link iff $(t, s)$ is an e-link.

Let $r_0, r_1, \ldots, r_l$ and $u_0, \ldots, u_m$ be e-chains. Then the concatenation $r_0, r_1, \ldots, r_l, u_1, \ldots, u_m$ is also an e-chain.

If the sequence $r_0, r_1, \ldots, r_l$ is an e-chain, then the reverse sequence $r_l, r_{l-1}, \ldots, r_0$ is also an e-chain.

We have the following proposition relating equalities and e-chains.

**Proposition 4.2 (E-chain).** Let $s$ and $t$ be ground terms. $s = t \in M(P^*)$ iff there is an e-chain for $(s, t)$.

It follows from the above results that $\equiv$ (as defined by $P^*$) is an equivalence relation: Let $s, t$, and $u$ be ground terms. (1) $t = t \in M(P^*)$; (2) if $s = t \in M(P^*)$, then $t = s \in M(P^*)$; (3) if $s = t$, $t = u \in M(P^*)$, then $s = u \in M(P^*)$.

From these results, it is straightforward to establish the following proposition by mathematical induction on the ordinal powers of the operator $T_2$ associated with a program $\mathcal{G}$. (Similar proofs are given in [3, 6].)
Proposition 4.3. Let $\Xi$ be the set of predicate symbols in $\mathcal{P} \cup \{ \text{ref, sym, tran} \} \cup \text{Pred}$. Then

$$\Xi_L(\mathcal{P}^*) = \Xi_L(\mathcal{P} \cup \{ \text{ref, sym, tran} \} \cup \text{Pred}).$$

Then it follows from Proposition 2.2 and Proposition 4.3 that $E$-chain subsumes the reflexivity, symmetry, transitivity, and predicate substitutivity axioms of equality.

Proposition 4.4. $\mathcal{P} \cup \{ \text{ref, sym, tran} \} \cup \text{Pred}$ and $\mathcal{P}^*$ are CAS-equivalent with respect to the set of predicate symbols in $\mathcal{P} \cup \{ \text{ref, sym, tran} \} \cup \text{Pred}$.

This completes our proof that $E$-Chain (as implemented by $\mathcal{P}^*$ in SLD resolution) is correct with respect to the equality axioms $\{ \text{ref, sym, tran} \} \cup \text{Pred}$. If $\text{Fun}$ is a subset of $\mathcal{P}$, then $\mathcal{P} \cup \text{Fun}$ and $\mathcal{P}^*$ are Cas-equivalent with respect to the predicate symbols in $\mathcal{P} \cup \text{Fun}$. In other words, $E$-Chain is correct (sound and complete) with respect to $\text{Fun}$.

5. SUMMARY AND FURTHER DEVELOPMENTS

We have introduced the notion of CAS-equivalent logic programs and established a proposition (2.2) that provides a least model criterion for two logic programs to be CAS-equivalent with respect to a set of predicate symbols. The least model approach has been illustrated by a proof of CAS-equivalence between $\text{REV}$ and $\text{REV1}$ with respect to the $\text{reverse}$ relation. $\text{REV}$ is an intuitive definition of $\text{reverse}$ in terms of $\text{append}$, while $\text{reverse}$ is defined by tail recursion in $\text{REV1}$. The power of the approach has further been illustrated by a proof of the correctness of $E$-Chain, a PROLOG metainterpreter for logic programs with equality.

The metainterpreter $E$-Chain, as it stands now, still has two major problems. First, we need to supplement $E$-Chain with the set $\text{Fun}$ of function substitutivity axioms. Secondly, $E$-Chain may still run into infinite loops. Methods for controlling infinite computations are presented in Chan [3]. Problems caused by the inclusion of $\text{Fun}$ are also considered there.

Transformation of logic programs to CAS-equivalent programs provides the foundations for semantic reduction, which is a general approach for reducing the semantics of extensions of SLD resolution to the semantics of SLD resolution. Given an extension $X$ of SLD resolution that builds in a set $\mathcal{A}$ of axioms, we show that there is a transformation $\tau$ on logic programs such that (1) there is a refutation from a program $\mathcal{P}$ and a goal $G$ according to $X$ iff there is an SLD refutation from the transformed program $\tau(\mathcal{P})$ and $G$; (2) $\tau(\mathcal{P})$ and $\mathcal{P} \cup \mathcal{A}$ are CAS-equivalent. It then follows that $X$ is sound and complete relative to $\mathcal{A}$. See Chan [3,5] for details.

The author would like to thank Dr. E. W. Elcock for carefully reading and providing detailed constructive comments on earlier drafts of the paper. Moreover, the author would like to thank Dr. W. Demopoulos, Dr. E. P. Stabler, Jr., and Mr. P. Hoddinott for rewarding discussions. The work has been supported in part under NSERC operating grant A2445 and in part of the IBM Corporation.
REFERENCES

7. Demopoulos, W., The Homogeneous Form of Logic Programs with Equality, Dept. of Philosophy, Univ. of Western Ontario, 1986.