COMPILATION OF SET TERMS
IN THE LOGIC DATA LANGUAGE (LDL)

Oded Shmueli†, *Shalom Tsur*, *Carlo Zaniolo*

MCC, Austin, Texas

ABSTRACT

We propose compilation methods for the efficient support of set term matching in Horn clause programs. Rather than using general-purpose set matching algorithms, we take the approach of formulating *at compile time* specialized computation plans that, by taking advantage of information available in the given rules, limit the number of alternatives explored. Our strategy relies on rewriting techniques to transform the problem into an "ordinary" Horn clause compilation problem, with minimal additional overhead. The execution cost of the rewritten rules is substantially lower than that of the original rules and the additional cost of compilation can thus be amortized over many executions.

1. Introduction

1.1. Overview

We propose compilation methods for supporting matching of set terms in Horn clause programs efficiently. This approach is the basis for the implementation of set terms as "first class" constructs in LDL. LDL is a Horn clause logic programming language (HCLPL) intended for data intensive knowledge-based applications [TZ86, BNRST87]. The language can handle complex data as treated in [AB87, KV84, KRS84, OO83] and it supports various extensions to pure HCLPLs such as negation, arithmetic, schema facility and sets. Since the language is intented for data intensive applications, it is assumed that only fully instantiated answers are of interest. This makes it possible to use an execution

† Current address: Department of Computer Science, Technion, Haifa, Israel 32000.

model that is based on matching and fixpoint operators, rather than full unification and SLD-resolution. Compile time techniques based on rule transformations are used to map LDL programs into this simpler execution model, resulting in an efficient implementation for the intended application domain [ZAN88]. Likewise, LDL's approach to set matching uses, at compile time, rewriting techniques that transform set matching into a sequence of ordinary matching problems. Thus, specialized computation plans are formulated that, by taking advantage of information available in the given rules, tailor the solution to the situation at hand and limit the number of alternatives and blind alleys explored at run time. As a result, the execution cost for rewritten rules is often substantially lower than that of the original rules and the additional cost of compilation is thus amortized over many query executions.

The techniques described in this paper are totally general and can be applied in any situation involving support of set terms via matching. In particular, they are applicable to the many cases where the execution is based on SLD-resolution, but matching can be used instead of full unification [MK85].

In this paper, we assume that set-objects are represented as terms whose main functor is *set_of*.[1] For example, the set {1,3,2} may be internally represented as *set_of(3,1,2)* (actually, it will be represented as *set_of(1,2,3)*). The characteristics of sets, in the mathematical sense, are captured by extending the notion of equality of such terms to account for the properties of commutativity and idempotence.

Example 1: Consider the rule:

$$\textbf{john\_friend(X)} \leftarrow \text{friends(set\_of(X,Y,john)), X} \neq \text{john, nice(X).}$$

Assume that the database[2] contains the following facts:

[1] As defined in Section 2.1, the word "term" refers to the elements of the Herbrand universe of the program.

[2] For notational convenience in defining the semantics, formally, the database is considered a part of the program. Our results hold for the case where the database is a separate entity provided the facts in the database are standardized (see section 3).

friends(set_of(john, jim, jack)).
nice(jim).
nice(jack).

The derived facts are john_friend(jim) and john_friend(jack).

The first answer comes from α={ X/jim, Y/jack}, and the fact that the set consisting of jim, jack, and john is the same as the set consisting of john, jim, and jack.

The second answer comes from β={ X/jack, Y/jim}, and the fact that the set consisting of jack, jim, and john is the same as the set consisting of john, jim, and jack. []

The basic mechanism used in the implementation of LDL is *matching*; i.e. the unification of a term with a ground term. In this paper, we concentrate on the mathematical principles underlying the efficient implementation of set matching. Versions of these methods tuned for maximum performance are employed in the actual implementation.

We assume that the reader is familiar with the basic notation of Logic Programming as presented, e.g., in [LLOY87]. For the purpose of this paper one can safely think of LDL as a pure HCLPL (with the distinguished, variable arity, functor *set_of*) whose semantics is defined using the T_P operator by "bottom-up" repeated "firing" until fixpoint [LLOY87] . The main difference between our T_P and the one in [LLOY87] is that instead of matching we use ci-matching[3] as defined below. In addition we use the variable arity functor *set_of*.

The *set_of* functor is used for the representation of traditional mathematical sets. As such, the order of arguments in a *set_of* term is immaterial; this is captured by the concept of **permutation**. Term t is a *permutation* of term s if t is obtained from s by a sequence of zero or more interchanges of arguments in *set_of* subterms of s. Likewise, repetitions of equal arguments should be ignored; this is captured by the concept of elementary compaction. Term t is an *elementary compaction* of term s if it is obtained from s by (i) locating a subterm A of s which has two identical arguments; say at positions i ,j

[3] In "ci-matching", c stands for commutative and i for idempotent.

such that $i < j$, and (ii) deleting the j'th argument from A. Terms t and s are *ci-equal*, denoted $t =_{ci} s$, if there is a sequence $t = t_1, \ldots, t_k = s$ such that for $i = 1, \ldots, k-1$, t_{i+1} is a permutation of t_i, or t_{i+1} is an elementary compaction of t_i, or t_i is an elementary compaction of t_{i+1}. Term t *ci-unifies* with term s if there exists a substitution α such that $t\alpha =_{ci} s\alpha$. In case s is ground and t ci-unifies with s, we say that t *ci-matches* s. Let us illustrate the above concepts.

Example 2: Consider again the rule:

john_friend(X) ← friends(set_of(X,Y,john)), X $\neq$ john, nice(X).

Assume that the database contains the following facts:

friends(set_of(jim, john)).
nice(jim).

The only derived fact is john_friend(jim).

There are three substitutions that map set_of(X, Y, john) to set_of(jim, john).

One is α= {X/jim, Y/jim} since set_of(jim, jim, john) $=_{ci}$ set_of(jim, john), it derives john_friend(jim).

The second is β= {X/jim, Y/john} since set_of(jim, john, john) $=_{ci}$ set_of(jim, john), it derives john_friend(jim).

The third is γ= {X/john, Y/jim} since set_of(john, jim, john) $=_{ci}$ set_of(jim, john); however, no fact is derived because of X $\neq$ john. []

If we modify the database in the above example to contain only the facts friends(set_of(john)) and nice(john), then the only applicable substitution is α= { X/john, Y/john } and set_of(john, john, john) $=_{ci}$ set_of(john). So, it is possible to specify a set containing three elements that is instantiated into a set containing (mathematically) one element. Again, no fact is derived because of X $\neq$ john.

We now illustrate the usefulness of the "i" in ci-matching. Suppose that out of experienced teams (*old_team*) we need a team where the expertise of engineer, pilot and medical doctor are represented. Then, we can use the rule:

ok_team(set_of(X,Y,Z)) ← old_team(set_of(X,Y,Z)),
engineer(X), pilot(Y), medical_doctor(Z).

Thus, *old_team (set_of (mark ,john))* would qualify as an *ok_team* if, for example, john is a medical doctor and mark is both a pilot and an engineer.

The meaning of an LDL program P is defined using ci-matching. In order to implement P efficiently we transform it into an equivalent program that employs only ordinary matching (two programs are equivalent when they produce the same set of answer tuples modulo ci-equality). Thus, the *set_of* terms in the transformed program are treated as ordinary terms, modulo a compaction and ordering operation which, when applied to newly derived facts, eliminates components of *set_of* terms so that no two subterms are ci-equal.

To transform a program P requiring ci-matching into one which requires ordinary matching, we expand the rules of P. The result for the rule in Example 1 is shown in Example 3 below. We introduce new rules called "funnel-up" rules[4], and use a short hand notation called multi-head-multi-body (MHMB) rules. In a MHMB rule, a comma is to be read as "and", and a semicolon as "or". For example the MHMB rule with two heads (a and b) and three bodies (1) c,d , (2) e,f , and (3) g,h

a,b ← c,d ; e,f ; g,h

represents the six rules:

a ← c,d   a ← e,f   a ← g,h   b ← c,d   b ← e,f   b ← g,h

So, a rule with m bodies and n heads represents $m \times n$ ordinary rules, one for each body-head combination.

[4]The term "funnel-up rule" stems from the role that these rules fulfill: they funnel data from one format (stored or already derived results) into another format, required by the structure of the original term in the body of a rule.

Example 3: Consider rule r; The rewritten rule is r'.

r: john_friend(X) $\leftarrow$ friends(set_of(X,Y,john)), X $\neq$ john, nice(X).

r': john_friend(X) $\leftarrow$ funnel_up_friends(set_of(X,Y,john)), X $\neq$ john, nice(X).

```
funnel_up_friends(set_of(Y,X,john)),
funnel_up_friends(set_of(X,Y,john))  ←   friends(set_of(john,Y,X));
                                          friends(set_of(Y,john,X));
                                          friends(set_of(Y,X,john)).

funnel_up_friends(set_of(X,X,john))  ←   friends(set_of(john,X));
                                          friends(set_of(X,john)).
funnel_up_friends(set_of(john,john,john))  ← friends(set_of(john)).
```

[]

In the case of Example 3, we have three MHMB rules, each supporting the ci-matching of the original term with instantiated *set_of* terms of cardinality three, two and one. The body of a rule checks for "generic" appearances of terms with a certain cardinality in the database. For example, in the second rule, *friends(set_of (john ,X))* and *friends(set_of (X ,john))* check for possible matches with a cardinality 2 instance. The heads of a MHMB rule "transmit" the found bindings to the original term in the original rule. In the second rule, bound values for

(1) *funnel_up_friend(set_of (X ,X ,john))*,

(2) *funnel_up_friend(set_of (X ,john ,john))*, and

(3) *funnel_up_friend(set_of (john ,X ,john))*

need to be transmitted, in order to account for commutativity (see discussion in Section 4). A closer inspection reveals that (1) and (2) will generate the same head tuples in r' and that (3) will violate $X \neq john$ in the original rule and hence (2) and (3) can be discarded. In fact, for the same reason, one can also delete the rule:

funnel_up_friends(set_of(john,john,john)) $\leftarrow$ friends(set_of(john)).

(Elimination of redundant rules is discussed in Section 6.)

The transformation result may seem bulky. However, run-time ci-matching on a per tuple basis is replaced as a result, with an optimized compile-time "unfolding" of the matching process. Our compile-time analysis eliminates blind alleys in ci-matching as well as redundant derivations; it also optimizes the ci-matching process in the context of the particular program (see section 4).

1.2. Relationship to unification in equational theories

Over the last two decades, unification in equational theories has constituted an extremely active research area; some, non exhaustive, references include [BHKSST88, BUTT86, HERO86,HERO86, FAGES87, LS76, RSSU79, STICK81, SIEK89]. In particular, several papers have studied unification for functions satisfying various combinations of following three axioms:

(1) (A : associativity) f(x,f(y,z))=f(f(x,y),z) for an associative function f.

(2) (C : commutativity) f(x,y)=f(y,x) for a commutative function f.

(3) (I : idempotence) f(x,x)=x for an idempotent function f.

Siekmann provides an excellent review on work in this area [SIEK89]. In particular, most of the proposed ACI unification algorithms transform the unification problem into that of solving a system of linear diophantine equations [FAGES87, LS76, STICK81, LC88]. These are of relevance to the problem at hand, inasmuch as the ACI framework has traditionally been viewed as a natural model for properties of sets. Unfortunately, this framework is not well-suited for our specific needs.

Consider for instance {a,b,c} and {X,Y}. According to LDL's well-defined semantics [BNRST87, NATS89] these are not unifiable. Indeed the variables in {X,Y} stand for set elements –although not necessarily distinct ones. Thus {X,Y} can only unify with sets containing one element or with sets containing two elements, while {a,b,c} has three elements. If we represent our two sets using an ACI function f, we then obtain the two terms f(a,f(b,c)) and f(X,Y). Hence, relative to ACI, these two terms are matchable;

with X=a and Y=f(b,c). Thus, Y plays the role of a subset, rather than an element as in the LDL semantics.

One can encode the LDL set semantics in the context of *ACI* unification using the following idea: each element is enclosed with a function symbol, say g, which has no axioms associated with it. Then sets will be represented as f(g(a),f(g(b),g(c))) and f(g(X),g(Y)), now matching with subsets is not possible as f and g do not match. Note that, in essence, a very limited form of associativity is used in this encoding.

The fact that we look at a simplified version of *ACI* matching, and that we do not reduce our problem to solving linear diophantine equations at run time, allows us to cast the matching problem in a natural way within LDL. This in turn opens the way for context related optimization of the matching process which is carried out at compile time.

Another technique for unification in equational theories is that of *narrowing* [DERSH87, FAY79, HULL80, RETY87]. Narrowing presents one possibility for constructing a universal unification algorithm [SIEK89]. Basically, narrowing transforms a pair of terms (the unificands) using a term rewriting system. If no rule is applicable, a substitution is applied to the terms so that a subterm of one of them is unified with a left hand side of a rewrite rule, the subterm is then replaced by the right hand side of the rule. Narrowing stops when, after a sequence of applications of substitutions and rewriting, the current two terms are syntactically unifiable.

To use narrowing as a unification technique, one uses a canonical (namely a confluent and terminating, i.e. noetherian) term rewriting system which represents the equational theory.[5] Obtaining such a system from a given set of axioms for the theory is called *completion*. The basic vehicle in this area is the Knuth-Bendix completion procedure [KB70]. Unfortunately, this procedure does not always succeed. Improving techniques for obtaining a confluent and terminating rewriting system is an area of active

[5] These concepts are defined in Section 2, see also [DERSH87].

research [BDP87, CHRIS89, DERSH87, PS81, JK83, JK86]. (Some of these approaches accept the fact that a theory has no completion as such, and provide rewriting rules that assume that a unification algorithm exists for a theory represented by a subset of the axioms.)

We have investigated the possibility of finding an equational theory with a completion for LDL sets. A natural representation for LDL sets was proposed by Jim Christian. This representation uses a binary function symbol, say g; it represents set_of(x) as g(x,nil), where nil is a constant that denotes the empty set. Then, set_of(x1, ... , xn) is represented as g(x1,g(x2, ... g(xn,nil))). The equational theory is:

(1) (commutativity) g(x,g(y,z)) = g(y,g(x,z))

(2) (idempotence) g(x,g(x,y)) = g(x,y)

Unfortunately, producing a confluent and terminating system for these equations appears to be beyond our analytical skills and the state-of-the art in completion algorithms. For instance, the HIPER system [Chris89], diverged on these equations, although it embeds most of the known completion techniques [PS81, JK86] along with several refinements.

There may be other ways of modeling the LDL set semantics in the framework of narrowing. However, the formalization proposed in this paper has some unique advantages which are not easily replicated by other approaches. For instance, in our approach, idempotence alone allows us to reduce a scheme $set\_of(\ldots,x,\ldots,x,\ldots)$ into $set\_of(\ldots,x,\ldots)$. This allows us to break the overall "narrowing" into two stages, one in which only commutativity is used and one in which only idempotence is used (Lemma 2.3). For example, in the equational framework above, we would need to use both the commutativity and idempotence properties of the function g to obtain a similar simplification involving non-adjacent terms.

In summary, we deal with a simpler theory than ACI. There are several similarities between our techniques and "standard" narrowing, but there are also substantial differences. **First,** we do not need a completion procedure and we utilize special properties of the (simpler) theory we work in. Second, we compile, ahead of time, the matching process that will be carried out at run time. Also, by casting the problem as a rule transformation problem, we take advantage of many optimization possibilities that would be difficult to detect by a general-purpose matching algorithm.

1.3. Paper organization

There are five sections. Section 2 discusses technical aspects of augmenting a HCLPL with the *set_of* functor. Section 3 presents two theorems. The first allows ci-matching to be substituted by i-matching, the second allows i-matching to be substituted for by ordinary matching. The rewriting transformation is presented in section 3. Optimization techniques are discussed in section 4. Section 5 concludes and mentions possibilities for future work.

2. Augmenting Logic Programming with CI-Matching

2.1. Horn clauses

A *term* t is defined inductively as (i) a constant, (ii) a variable, (iii) a formula of the form $f(a_1, \ldots, a_n)$ where f is a function symbol and, for $i=1, \ldots, n$, a_i is a term which is called the *argument of t of index i*. The *height* of a term t, denoted $height(t)$, is defined inductively thus: the height of a constant is zero; the height of $f(t_1, \ldots, t_n)$ is $1+\max\{height(t_1), \ldots, height(t_n)\}$.

A rule is a formula of the form:

$$A \leftarrow B_1, \ldots, B_n$$

where A and each B_i, $0 \leq i \leq n$, are *literals*, i.e. a predicate symbol applied to as many terms as indicated by its arity. Let $arity(t)$ denote the arity of literal or term t.

In the rest of the paper, we will explicitly define various syntactic notions on terms (e.g., ci-equality and ci-matching). We also implicitly extend the same notions to literals, which have the same syntactic form as terms. Thus we will use the same terminology for both terms and literals.

A *substitution* is a set of pairs $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$ where $X_1, \ldots, X_n$ are distinct variables and $t_1, \ldots, t_n$ are terms. Then $t\theta$, the *instance of term t by θ*, is the expression obtained from t by simultaneously replacing, for $i=1, \ldots, n$, each occurrence of the variable X_i, in t by the term t_i. The *composition* $\theta\sigma$ of two substitutions $\theta=\{X_1/t_1, \ldots, X_m/t_m\}$ and $\sigma=\{Y_1/s_1, \ldots, Y_n/s_n\}$ is the substitution obtained from the set

$$\{X_1/t_1\sigma, \ldots, X_m/t_m\sigma, Y_1/s_1, \ldots, Y_n/s_n\}$$

by deleting every binding $X_i/t_i\sigma$ for which $X_i=t_i\sigma$ and each binding Y_j/s_j for which $Y_j \in \{X_1, \cdots, X_m\}$. A substitution θ is *a generalization* of a substitution δ if there exists a substitution α such that $\delta = \theta\alpha$.

A substitution $\theta=\{X_1/t_1, \ldots, X_n/t_n\}$ where $t_1, \ldots, t_n$ are all *ground*, i.e., contain no variables, is a *binding*. A term t_1 is said to be *more general than (or a generalization of)* a term t_2 when there exists a substitution θ such that $t_1\theta = t_2$; in that case t_2 is a *restriction* of t_1; if t_2 is ground then t_2 is an *instantiation* of t_1. If two terms are each a generalization of the other, then they differ only by variable renaming and they are said to be *variants* of each other.

A substitution θ is said to *unify (or, to be a unifier for)* two terms t_1 and t_2 if $t_1\theta = t_2\theta$; then we also say that the unification equation $t_1 = t_2$ is *satisfiable* and θ is a *solution* for that equation. A set S of unification equations is *satisfiable* if there exists a substitution θ such that θ is a solution for each equation in S. From the existence of a most general unifier of two terms [LLOY87], it follows that:

Proposition 2.1: Given a satisfiable finite set of unification equations U, there exists a solution θ which is a generalization of every solution for U. []

A most general solution for U will be called a *most general unifier* (mgu) for U. So far, our concepts of equality and unification are the standard ones where two terms are equal iff they are (syntactically) identical, and are unifiable iff the unification equation for them is satisfiable.

2.2. CI-matching

We assume that *set_of* is a distinguished function symbol that models mathematical sets; as such, it does not have a fixed arity. With zero arity, i.e. *set_of* ( ), it represents the empty set. With non-zero arity, i.e. *set_of* $(a_1, \ldots, a_n)$, it represents the set whose elements are $a_1, \ldots, a_n$ (not necessarily distinct). These intuitive notions are captured formally as relations on terms.

The binary relation (on terms s, t) *reduction by idempotence*, denoted $t \Longrightarrow_i s$, holds when s is t with the exception that a subterm t_1 of t, $t_1 = set\_of(x_1, \ldots, x_i, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_n)$, such that $x_i = x_j$, $i < j$, is modified by deleting x_j to obtain $s_1 = set\_of(x_1, \ldots, x_i, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n)$ in s. We also say that s is obtained from t by an *elementary compaction step*. Observe that $t \Longrightarrow_i s$ does not imply $s \Longrightarrow_i t$.

The binary relation (on terms s, t) *reduction by commutativity*, denoted $t \Longrightarrow_c s$, holds when s is t with the exception that a subterm t_1 of t, $t_1 = set\_of(x_1, \ldots, x_i, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_n)$, is modified by exchanging arguments x_i and x_j to obtain $s_1 = set\_of(x_1, \ldots, x_j, \ldots, x_{j-1}, x_i, x_{j+1}, \ldots, x_n)$ in s. We also say that s is obtained from t by a *permutation step*. Observe that $t \Longrightarrow_c s$ iff $s \Longrightarrow_c t$.

The binary relation (on terms s, t) *reduction by commutativity and idempotence*, denoted $t \Longrightarrow_{ci} s$, holds if either (i) $t \Longrightarrow_i s$; or (ii) $t \Longrightarrow_c s$.

Each of $\Longrightarrow_c$, $\Longrightarrow_i$ and $\Longrightarrow_{ci}$ defines a binary relation on terms which may contain the set_of distinguished functor. Let $\overset{*}{\Longrightarrow}_c$, $\overset{*}{\Longrightarrow}_i$ and $\overset{*}{\Longrightarrow}_{ci}$ denote the transitive and reflexive closure of $\Longrightarrow_c$, $\Longrightarrow_i$ and $\Longrightarrow_{ci}$, respectively. Also, let $=_i$, $=_c$ and $=_{ci}$ denote the transitive, reflexive and symmetric closure of $\Longrightarrow_c$, $\Longrightarrow_i$ and $\Longrightarrow_{ci}$, respectively.

Properties of *reductions*, binary relations on terms, have been extensively investigated [HUET80]. A reduction R is *confluent* if, whenever tR^*t_1 and tR^*t_2 then there exists t_3 such that $t_1R^*t_3$ and $t_2R^*t_3$, where R^* is the reflexive and transitive closure of R. It can be shown that both $\Longrightarrow_c$ and $\Longrightarrow_i$ are confluent. This is straightforward for $\Longrightarrow_c$, it requires an induction on the height of a term for $\Longrightarrow_i$.

Another important property of reductions is termination. R *terminates* if there is no infinite sequence $tRt_1Rt_2 \cdots$. If $tRt_1Rt_2 \cdots Rt_m$ is such that there is no s such that t_mRs, then t_m is called a *normal form for t*. It can be easily seen that while $\Longrightarrow_i$ is terminating, $\Longrightarrow_c$, and hence also $\Longrightarrow_{ci}$, is non-terminating. We note that a relevant concept is that of a *term rewriting system*, such a system is a finite set of *rewrite rules* of the form $l \longrightarrow r$, where l and r are terms. Each of $\Longrightarrow_c$, $\Longrightarrow_i$ and $\Longrightarrow_{ci}$ can be thought of as defining a "generalized" rewrite rule, "generalized" because the objects related by the rules are specified by "patterns" rather than by terms (manifested by the use of " $\cdots$ " in defining $\Longrightarrow_c$ and $\Longrightarrow_i$).

Next, we extend equality based unification and matching. A substitution θ *i-unifies, c-unifies, ci-unifies* terms t_1 and t_2 if $t_1\theta =_i t_2\theta$, $t_1\theta =_c t_2\theta$, $t_1\theta =_{ci} t_2\theta$, respectively. When t_2 is ground the word *unifies* is replaced by *matches*; we then speak of *i-matching, c-matching* and *ci-matching*.

Term t is *compact* if it contains no *set_of* subterm with two syntactically identical arguments. Equivalently, t is compact if $t \overset{*}{\Longrightarrow}_i s$ implies $t{=}s$. For example,

$$f\,(22, set\_of\,(1,2,3), 22)$$

is compact, **while**

$$f\,(22, set\_of\,(1,2,1,3), 22)$$

is not compact. Term t is *strongly compact* if for all terms s such that $t \overset{*}{\Longrightarrow}_c s$, s is compact; intuitively, one cannot permute the arguments of *set_of* subterms of t and produce two identical ones. For example,

$$set\_of\,(set\_of\,(X,a), set\_of\,(a,X))$$

is compact but not strongly compact since

$$set\_of\,(set\_of\,(X,a),set\_of\,(a,X)) \Longrightarrow_c set\_of\,(set\_of\,(a,X),set\_of\,(a,X)).$$

Term s is a *strong compact form* for term t if s is strongly compact and $t =_{ci} s$. Strong compact forms are not unique in general. But, it can be shown that if t_1 and t_2 are both strong compact forms of t then $t_1 =_c t_2$.

A *substitution*, $\{X_1/t_1, \ldots, X_n/t_n\}$ is called *compact, or strongly compact,* when each t_i, $1 \leq 1 \leq n$, is compact or strongly compact, respectively.

Since $\Longrightarrow_i$ is a terminating reduction, starting with a term t and repeatedly carrying out rewriting applications one must reach a normal form. Clearly, this normal form is compact. Furthermore, since $\Longrightarrow_i$ is confluent, this normal form is unique [HUET80]. Thus, given a t, there exists a unique t' such that $t \overset{*}{\Longrightarrow}_i t'$ and t' is compact; t' will be called the *compact form* of t, denoted $com(t)$.

In this paper, we will use derivations $\overset{*}{\Longrightarrow}_i$ where, informally, the inner terms are reduced before the outer ones. That is to say that $x_i = x_j$ being compact is a precondition for reducing $set\_of\,(x_1, \ldots, x_i, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_n)$ to $set\_of\,(x_1, \ldots, x_i, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n)$. Reduction sequences satisfying this constraint will be called *bottom-to-top* compactions.

The proof of the following Lemma follows directly from the definitions.

Lemma 2.1: If I is compact and $t \overset{*}{\Longrightarrow}_i I$ then I can be obtained from t via a bottom-to-top compaction.

Clearly, **Lemma 2.1** holds when I is strongly compact.

The following Lemma states that if $t =_i I$ and I is compact then there is a sequence of duplicate elimination operations on *set_of* subterms of t that leads from t to I. Note that this is not always the case if I is not compact.

Lemma 2.2: Let I be compact. $t =_i I$ iff $t \overset{*}{\Longrightarrow}_i I$.

Proof: (if) Obvious. (only if) By Lemma 2.1 in [HUET80], $t =_i I$ iff there exists s such that $t \overset{*}{\Longrightarrow}_i s$ and $I \overset{*}{\Longrightarrow}_i s$. But, since I is compact, $s = I$, i.e. $t \overset{*}{\Longrightarrow}_i I$. □

Clearly, Lemma 2.2 holds when I is strongly compact.

Next, we show that if I is strongly compact and $t =_{ci} I$ then I can be obtained from t by first permuting some arguments of some *set_of* subterms of t and then performing a sequence of duplicate elimination operations from *set_of* subterms.

Lemma 2.3: Let I be strongly compact. $t =_{ci} I$ iff there exists w such that $t \overset{*}{\Longrightarrow}_c w \overset{*}{\Longrightarrow}_i I$.

Proof: (if) Obvious. (only if) $t =_{ci} I$ implies there exists a sequence $t = T_0, \ldots, T_k = I$ such that for $j = 0, \ldots, k-1$, $T_j \Longrightarrow_c T_{j+1}$, or $T_j \Longrightarrow_i T_{j+1}$ or $T_{j+1} \Longrightarrow_i T_j$. The proof is by induction on k, the length of the sequence.

Basis: ($k = 1$) Obvious.

Induction: ($k > 1$).

Case (i) $T_0 \Longrightarrow_c T_1$, obvious by the induction hypothesis.

Case (ii) $T_0 \Longrightarrow_i T_1$. We show that, intuitively, this "deletion" can be delayed. Suppose this step treats a *set_of* subterm v of t and it deletes argument B of v because of a lower index argument A in v. By hypothesis, there is a sequence S of steps establishing $T_1 \overset{*}{\Longrightarrow}_c w' \overset{*}{\Longrightarrow}_i I$. Modify S to create a new sequence S' that "operates" on T_0. For each permutation step in $T_1 \overset{*}{\Longrightarrow}_c w'$ which modifies a subterm of the term originating in A, add a new step that does the same to the corresponding subterm of the term originating in B. Thus, in the resulting w'', the terms A', B' which originate in A, B, respectively, are identical. Hence, before the sequence corresponding to $w' \overset{*}{\Longrightarrow}_i I$, add a step $w'' \Longrightarrow_i w'$. The resulting overall sequence proves the claim with $w = w''$.

Case (iii) $T_1 \Longrightarrow_i T_0$, by adding a subterm B because of a subterm A. We show that, intuitively, this "addition" is unnecessary and will induce a corresponding deletion later on; thus we can ignore it and its related operations altogether and end up with the same final term. By hypothesis, there exists w' such that $T_1 \stackrel{*}{==>}_c w' \stackrel{*}{==>}_i I$. Furthermore, there is a sequence of bottom-to-top compaction steps establishing $w' \stackrel{*}{==>}_i I$ by Lemma 2.2. Let A' and B' be the terms in w' originating in A, B, respectively. Each step in establishing $w' \stackrel{*}{==>}_i I$ is applied to a *set_of* subterm such that all its arguments are compact.

There must be a step deleting (a subsequent version of) A' or B'. Otherwise, both "survive" contradicting I strongly compact. If (a subsequent version of) B' is deleted, we can modify the whole sequence by not adding it to begin with and deleting any step referring to a subterm of the term originating in B. Therefore we have a shorter sequence, by the induction hypothesis we are done.

If no subsequent version of B' is deleted, we can produce a modified sequence establishing $t =_{ci} I$, of length less than or equal to k, which, instead of "adding" B, starts with permuting A to B's position (of addition), and deleting any reference to A or subterms thereof in the original sequence. So, this is Case (i). []

2.3. The standard representation of facts

A *fact* is a ground term. We start by defining a total order on facts.

(1) There is a total order on constants and function symbols (e.g., lexicographic ordering).

(2) If $t=f(t_1, \ldots, t_n)$ and $s=g(s_1, \ldots, s_m)$ and f precedes g, then t precedes s.

(3) If $t=f(t_1, \ldots, t_n)$ and $s=f(s_1, \ldots, s_m)$ then t precedes s if there exists $i \leq max(m,n)$ such that s and t are equal on positions $1, \ldots, i-1$ and either t_i precedes s_i or there is no position i in t.

A fact is in *sorted form* if in each *set_of* subterm of the fact, the arguments are in sorted order according to the total order defined above on facts.

We make the following two assumptions concerning stored facts. First, *facts are always in strongly compact form*. Second, *facts are always in sorted form* (see above). These two assumptions together constitute *the standard representation assumption*. A fact obeying this assumption is said to be *standard*. A binding $\theta=\{X_1/T_1, \ldots, X_k/T_k\}$ is *standard* if for $i=1,\ldots,k$, T_i is standard (recall that for a binding, all T_i are ground).

Given a fact t, the standard form of t, denoted $standard(t)$, is obtained from t by sorting each *set_of* subterm of t and eliminating duplicates in such a way that a subterm is handled only after all its *set_of* subterms have been handled. It can be shown that $standard(t)$ is unique and that $t \overset{*}{==>}_{ci} standard(t)$ which implies $t =_{ci} standard(t)$.

To illustrate the importance of the standard representation assumption, let us assume that we admit in the database the pair of facts *p(set_of(1,2))* and *q(set_of(2,1))* thereby violating this assumption. Then, by the semantics of sets, the conjunct *p(X),q(X)* must succeed, but that cannot be accomplished with ordinary matching – a direct contradiction to our basic tenets. Fortunately, this problem can be solved by assuming that database facts obey the standard representation assumption as defined above.

2.4. Semantics

The semantics of LDL sets is defined formally in [BNRST87]. Here we limit attention to a subset of LDL that is comprised of Horn clauses, the distinguished function symbol *set_of*, and two built-in predicate symbols = and $\neq$ which are of arity two and are written in infix notation. As mentioned, for simplicity, we view the database as part of the program. Binding θ ***satisfies*** rule $h \leftarrow t_1, \ldots, t_n$ in a set of facts S, if (i) it assigns (ground) terms to all the variables appearing in the rule; and (ii) for $i=1, \ldots, n$, either t_i is $s_1{=}s_2$ and $s_1\theta =_{ci} s_2\theta$, or t_i is $s_1{\neq}s_2$ and $s_1\theta \neq_{ci} s_2\theta$, or there exists $s_i \in S$ such that $t_i\theta =_{ci} s_i$. The ***model*** of a program P, denoted $M(P)$ is defined thus. Let $M_0=\emptyset$. For $i>0$ define:

$$M_i=M_{i-1} \cup \{\ h\theta \mid \text{binding } \theta \text{ satisfies } r: h \leftarrow t_1, \ldots, t_n \text{ in } M_{i-1}\}.$$

$$M(P)=\bigcup_{i=0}^{\infty} M_i .$$

In the sequel we shall refine components in both the model and rule satisfaction definitions. Our goal will be to show that each modification "preserves" the model.

Preservation is captured formally as follows. Two sets of facts S and T are *ci-equivalent*, denoted $S =_{ci} T$, if for all $s \in S$ there exists $t \in T$ such that $s =_{ci} t$ and vice versa.

We show that if θ is restricted to be standard, the resulting set of facts is $=_{ci}$ to $M(P)$.

Lemma 2.4: Let $M'(P)$ be defined like $M(P)$ except that M_i' is defined as:

$$M_i' = M_{i-1}' \cup \{\ h\theta \mid \text{standard binding } \theta \text{ satisfies } r: h \leftarrow t_1, \ldots, t_n \text{ in } M_{i-1}'\}.$$

Then, $M'(P) =_{ci} M(P)$.

Proof: Certainly $M'(P) \subseteq M(P)$. Each fact in the $\bigcup_{i=0}^{\infty} M_i$ is added by *some* S_i. Therefore, it suffices to show that for each $h\theta$ added to M_{i-1} to form M_i, there exists $h\overline{\theta} =_{ci} h\theta$ added to M_i' to form M_i, where θ is standard and the prime indicates construction of $M(P)$ under the Claim's restrictions. The proof is by induction on i, the basis, $i=0$, is obvious as $M'(P)=M(P)=\emptyset$. Suppose $\theta=\{\ X_1/T_1, \ldots, X_k/T_k\}$. Consider $h \leftarrow t_1, \ldots, t_n$ which is satisfied by θ in M_{i-1}. The argument for t_i of the form $a=b$ or of the form $a \neq b$ is similar to the one that follows; so, w.l.o.g. the predicate symbol of t_i is not $=$ or $\neq$, $1 \leq i \leq n$.

Therefore, for $j=1, \ldots, n$ there exists a fact $a_j \in M_i - 1$ such that $t_j\theta =_{ci} a_j$. By hypothesis, for $j=1, \ldots, n$, there exist $b_j \in M_{i-1}'$ such that $a_j =_{ci} b_j$. Let $\overline{\theta}=\{X_1/\mathit{standard}(T_1), \ldots, X_k/\mathit{standard}(T_k)\}$. By construction, for $j=1, \ldots, n$, $t_j\overline{\theta} =_{ci} t_j\theta =_{ci} a_j =_{ci} b_j$. Thus, in forming M_i' $h\overline{\theta}$ is added; furthermore $h\overline{\theta} =_{ci} h\theta$. []

The set of facts obtained when in addition each derived fact is standardized before being added to the model, is also $=_{ci}$ to $M(P)$.

Lemma 2.5: Let $M''(P)$ be defined like $M(P)$ except that M_i'' is defined as: $M_i''=M_{i-1}'' \cup \{$ $\mathit{standard}(h\theta) \mid$ standard binding θ satisfies $r: h \leftarrow t_1, \ldots, t_n$ in $M_{i-1}''\}$. Then, $M''(P) =_{ci} M(P)$.

Proof: It suffices to show that for $i \geq 0$, $M_i =_{ci} M_i^{''}$ where the double prime indicates construction of $M(P)$ under the Claim's restrictions and the single prime indicates construction under the restrictions of Lemma 2.4. The proof is by induction on i, the basis ($i=0$) is obvious. For the induction step it suffices to show that a fact t is added to form M_i' iff a standard form $t^{''}$ is added to form $M_i^{''}$ where $t =_{ci} t^{''}$.

(=>) Consider $h \leftarrow t_1, \ldots, t_n$ which is satisfied by θ in M_{i-1}'. Therefore, for $j=1, \ldots, n$ there exists a standard form fact $a_j \in M_i - 1$ such that $t_j\theta =_{ci} a_j$. By hypothesis, for $j=1, \ldots, n$, there exist $b_j \in M_{i-1}^{''}$ such that $a_j =_{ci} b_j$. Thus, for $j=1, \ldots, n$, $t_j\theta =_{ci} a_j =_{ci} b_j$. It follows that *standard*$(h\theta)$ is added to form $M_i^{''}$. Since *standard*$(t) =_{ci} t$, the claim follows. (<=) A similar argument applies in this direction. []

Let P be a program and q a literal. The *correct result* for *query* q *against* P is

$$\{q\theta \mid \text{there exist } \theta,\ s \in M(P) \text{ such that } q\theta =_{ci} s\}$$

It can be shown that if $M(P)$ above is replaced with any set S such that $S =_{ci} M(P)$, then the same set of result facts is obtained. This indicates that we deal with mathematically identical sets of complex objects. In practice, a set of answers is most probably infinite, e.g. if $\theta = \{ X_1/set\_of(1) \}$ then $\theta = \{ X_1/set\_of(1,1) \}$ will do as well as $\theta = \{ X_1/set\_of(1,1,1) \}$ and so on. So, in practice, one might be satisfied with any set that is $=_{ci}$ to the answer set defined herein.

Using Lemma 2.4 and Lemma 2.5 we obtain:

Theorem 1: Suppose in the definition of $M(P)$ each added fact is standardized, and at least all **standard substitutions** are considered (and perhaps some non-standard ones are considered as well); let $M_1(P)$ be the resulting model. Then, $M_1(P) =_{ci} M(P)$. []

Intuitively, Theorem 1 states that if generated facts are standardized, all standard substitutions are considered, and some additional substitutions are considered as well, the result is still $=_{ci}$ to $M(P)$.

2.5. The C-decomposition Theorem

The following Theorem is the basis of the first step in program rewriting, replacing ci-matching with i-matching by considering all permutations of a term for i-matching. This depends on being able to commute substitutions and permutations.

Theorem 2: Let I be a standard fact and θ a standard substitution; $t\theta =_{ci} I$ iff there exist t_1 such that $t =_c t_1$ and $t_1\theta =_i I$.

Proof:

(<=) If $t =_c t_1$ then $t\theta =_c t_1\theta$ with the same $\overset{\bullet}{==>}_c$ sequence. Since $t_1\theta =_i I$, $t\theta =_{ci} I$.

(=>) Let $s = t\theta$, by Lemma 2.3 $s =_{ci} I$ implies there exists s_1 such that $s =_c s_1$ and $s_1 \overset{\bullet}{==>}_i I$ where by Lemma 2.1 $s_1 \overset{\bullet}{==>}_i I$ can be shown via a bottom-to-top compaction. Consider variable X_1 which is replaced by θ with term T_1. Let T_{11} be the subterm in s_1 corresponding to an occurrence of T_1 in s. Since I is standard T_{11} must be standard as well. This is because T_{11} cannot be compacted any more and some trace of it, i.e. an equal subterm if T_{11} is deleted in $s_1 \overset{\bullet}{==>}_i I$, must equal a subterm of I. Thus $T_1 = T_{11}$. A similar argument holds for all X_i. In other words, the $\overset{\bullet}{==>}_c$ steps leave each subterm originating in a θ replaced variable as is; it follows that all $\overset{\bullet}{==>}_c$ steps operating on such subterms may be dropped without affecting the outcome. Let t_1 be the term obtained from t by the remaining steps. Observe that $t_1\theta = s_1$. It follows that there is $t_1 =_c t$ such that $t_1\theta = s_1 =_i I$. □

Note that the above theorem would not hold if θ is not required to be standard. For example, if $t = set\_of(X,Y)$, $I = set\_of(set\_of(1,2))$ and $\theta = \{X/set\_of(1,2), Y/set\_of(2,1)\}$, then $t\theta =_i I$, but there is no t_1 such that $t =_c t_1$ and $t_1\theta =_i I$.

2.6. The I-decomposition Theorem

The second main step in the rewriting presented in this paper is replacing i-matching with ordinary matching. This is done by determining a priori the possible identification of subterms that could be made by run-time substitutions. Essentially, this tantamounts to considering each possible bottom-to-top compaction and solving a set of (ordinary) unification equations implied by the bottom-to-top compaction. We need some machinery to carry out this task.

We need a mechanism to refer to subterm positions independent of their "current" content, this is analogous to the distinction between a variable and its content. Any subterm of a term t can uniquely be identified by its *term address*, defined as follows:

(i) γ is a term address whose content is the whole term t,

(ii) if I is the term address in t whose content is the subterm $f(t_1, \ldots, t_n)$ then $I.j$, $1 \leq j \leq n$, is a term address in t whose content is t_j.

We use $t.I$ to denote the subterm of t whose address is I (e.g., $t.\gamma = t$). For example, if $t = f(g(s_1,s_2),h(X))$ then $t.\gamma.2 = h(X)$ and $t.\gamma.1 = g(s_1,s_2)$ and $t.\gamma.1.2 = s_2$ in t.

An $E-entry$ on term t is of the form $I.i = I.j$ where I is the address of a *set_of* subterm of t, $i < j$ and i and j are addresses of arguments of $t.I$. For example, let $t = f(set\_of(a,X), set\_of(b,Y,b), X)$ then $\gamma.2.1 = \gamma.2.3$ is an E-entry on t. Intuitively, an E-entry means that during a bottom-to-top compaction on $t\theta$ for some θ the subterms at these addresses will be equal. In the last example, indeed $b = t.\gamma.2.1 = t.\gamma.2.3 = b$ and a bottom-to-top compaction could delete the second b. As another example consider the E-entry $\gamma.1.1 = \gamma.1.2$. This E-entry means that during a bottom-to-top compaction on $t\theta$, for some θ, the subterms originating with a and X will be equal. This implies a unification equation, namely $a = X$.

An ***E-sequence*** E on t is a sequence of E-entries on t such that for all $A = B$ appearing in the sequence no address of the form $A.\alpha$ or of the form $B.\alpha$ appears later on in the sequence. Intuitively, an E-sequence depicts a bottom-to-top compaction on $t\theta$ for some θ. Continuing the example, $E = (\gamma.2.1 = \gamma.2.3,\ \gamma.1.1 = \gamma.1.2\ )$ is an E-sequence on t. Observe that an E-sequence defines a sequence of unification equations and also a "final result" and an mgu, these are formally defined below.

We adopt the convention that for an equation $t_1=t_2$, the mgu is produced by the unification algorithm in [LLOY87]. Thus, such an mgu only assigns to variables that appear in t_1 or t_2, and furthermore, it only assigns terms built out of the constants, function symbols and variables appearing in t_1 or t_2. In our example, the final compacted result is $f\,(set\_of\,(a),set\_of\,(b\,,Y),a))$ and the mgu is $\{X/a\}$.

In general, an E-sequence $E=E_1, \ldots, E_n$ defines a set $Q(E)$ of unification equations and a term obtained from t denoted $E(t)$ which are obtained using algorithm sequence_application below.

```
algorithm sequence_application(E, t);
begin
  Q:=∅;
  s:=t;
  for k:=1 to n do
    begin
      let E_k be 'I.i =I.j';
      if I.i or I.j is not an address in s then
 abort;
      add to Q the equation s.I.i =s.I.j;
          /* the added equation is an equation between real terms not
 addresses */
      update s by deleting subterm s.I.j
    end;
  let E(t) be s;
  let Q(E) be Q
end.
```

An E-sequence is *executable* in t if the above algorithm does not abort on input t and E. Intuitively, if an E-sequence is not executable it definitely does not describe a bottom-to-top compaction on t. However, even if an E-sequence is executable it does not necessarily describe a bottom-to-top compaction since the unification equations may not be satisfiable. Furthermore, even if the unification equations are satisfiable, say with mgu ω, it still might be that $E(t)\omega$ is not strongly compact, and hence cannot depict the application of a binding followed by a bottom-to-top compaction ending up with a standard fact.

An E-sequence E is *satisfiable* in t if it is executable in t and $Q(E)$ is satisfiable. If E is satisfiable in t with ω an mgu for $Q(E)$, and $E(t)\omega$ is strongly compact, then $E(t)\omega$ is called the *generic term for t defined by E*. If E defines a generic term g for t then g is a variant of any other generic term defined by E for t.[6]

A substitution $\{X_1/T1, \cdots, X_n/Tn\}$ is *strongly compact* if for $1 \leq i \leq n$, T_i is a strongly compact term.

Claim 1: Let g be a generic term for t defined by an E-sequence E with mgu ω for $Q(E)$. Then, (i) $t\omega =_i g$, and (ii) ω is strongly compact.

Proof: (i) Since ω satisfies $Q(E)$ one can carry on $t\omega$ the elementary compaction steps defined by E. These operations relate to addresses in t. We thus end up with $E(t)\omega = g$.

(ii) Consider the elimination steps in producing $E(t)$. The result is $E(t)$ such that $E(t)\omega$ is strongly compact. Consider now $t\omega$. Suppose the elimination steps in forming $E(t)$ are applied, in the same order, to the same term addresses in $t\omega$. Clearly, the result is $E(t)\omega$. Suppose now that ω is not strongly compact, i.e. to some variable X it assigns a term T which is not strongly compact. Consider the subterm T corresponding to an X occurrence in t which appears in $t\omega$. No elimination step has an address within T, these addresses did not exist in t. Each elimination step, eliminates one of two equal terms, as ω is a solution to the unification equations. Thus, the final result, $E(t)\omega$ must contain a subterm equal to T. But since T is not strongly compact, this contradicts $E(t)\omega$ being strongly compact. []

Theorem 3: Let t be a term and I be a standard fact. The following two statements are equivalent:

(1) There exists a substitution θ such that $t\theta =_i I$ and θ is standard.

[6] In the actual implementation, we are using improved algorithms for obtaining the set of generic terms associated with a given term. While these algorithms are heuristically effective, their worst case behavior remains exponential in the size of t. This is not surprising as set matching is NP-hard [KN86].

(2) **There exist a substitution** δ and an E-sequence E with ω an mgu for $Q(E)$, defining a **generic term** $g=E(T)\omega$ for t, and such that $g\delta=I$ and $\omega\delta$ is standard.

Proof:

(1)<=(2) By above Claim $t\omega=_i g$, hence $t\omega\delta=_i g\delta=I$. Thus, $t\omega\delta=_i I$ and $\omega\delta$ is a standard substitution as required, i.e. $\theta=\omega\delta$ satisfies (1).

(1)=>(2) By Lemma 2.1 there is a bottom-to-top compaction of $t\theta$ into I. Let E be the E-sequence induced by this bottom-to-top compaction. Consider an E-entry $I.i=I.j$ in E. We claim that the subterm at address A has **not** originated in a T such that X_i/T is a pair in θ, this is because such a T is ground and strongly compact. Thus E is actually an E-sequence on t as well. Let $Q(E)$ and $E(t)$ be the set of unification equations and term obtained by algorithm sequence_application on input E and t. First, $Q(E)$ is satisfiable, simply observe that θ is a solution. Let ω be an mgu for $Q(E)$; thus there exists δ such that $\theta=\omega\delta$ [LLOY87]. Second, $E(t)\theta=I$. This is because E is an E-sequence on t and hence $I=E(t\theta)=E(t)\theta$. Third, $E(t)\omega$ is strongly compact. Suppose not; since $E(t)\omega$ is not strongly compact, certainly $E(t)\omega\delta$ is not strongly compact, but $E(t)\omega\delta=E(t)\theta=I$, I is strongly compact and hence a contradiction. Thus, $g=E(t)\omega$ is a generic term for t. Also, $g\delta=E(t)\omega\delta=E(t)\theta=I$. []

3. The Rewriting Transformation

Let $G(t)$ denote the set of all possible pairs (g,ω) such that $g=E(t)\omega$ is a generic term for t induced by some E-sequence E. In this section we will use $G(t)$ to transform the original **rules,** which assume ci-matching, into an equivalent set of rules that use only ordinary **matching.** Note that $G(t)$ is finite as there are only finitely many E-sequences for a term t.

3.1. The first step

We now explain the transformation. A rule r of the form $head \leftarrow t_1, \ldots, t_n$ where, w.l.o.g., t_1 contains *set_of* subterms is transformed into a rule r' of the form:

$head \leftarrow funnel\_up\_t_1, t_2, \ldots, t_n.$

and a set of *permutation rules*:

$funnel\_up\_t_1 \leftarrow permute\_1\_t_1.$

$\ldots$

$funnel\_up\_t_1 \leftarrow permute\_m\_t_1.$

where $permute\_1\_t_1, \ldots, permute\_m\_t_1$ are all the permutations of term t_1. Each such permutation is obtained from t by exchanging positions of arguments of some set_of sub-terms of t. The number of such permutations is obviously finite.

For the rule in example 2 we get:

john_friend(X) ← funnel_up_friends(set_of(X,Y,john)), X ≠ john, nice(X).

funnel_up_friends(set_of(X,Y,john)) ← friends(set_of(X,Y,john)).
funnel_up_friends(set_of(X,Y,john)) ← friends(set_of(X,john,Y)).
funnel_up_friends(set_of(X,Y,john)) ← friends(set_of(Y,X,john)).
funnel_up_friends(set_of(X,Y,john)) ← friends(set_of(Y,john,X)).
funnel_up_friends(set_of(X,Y,john)) ← friends(set_of(john,X,Y)).
funnel_up_friends(set_of(X,Y,john)) ← friends(set_of(john,Y,X)).

Let $P+funnel$ be the program resulting by transforming rule r in P as above. For a set of facts S, let S/P be the set of facts in S whose predicate symbol appears in P. Refine the notion of rule satisfaction as follows. Binding θ *satisfies* rule $h \leftarrow t_1, \ldots, t_n$ in a set of facts S, if all the variables appearing in the rule are assigned by θ and for $i=1, \ldots, n$, (i) if t_i is of the form $a=b$ then $a\theta =_{ci} b\theta$, (ii) if t_i is of the form $a \neq b$ then $a\theta \neq_{ci} b\theta$; otherwise, there exists $s_i \in S$ such that (a) $t_i\theta = s_i$ if t_i is a funnel_up literal, (b) $t_i\theta =_i s_i$ if t_i is a permute_j literal, and otherwise $t_i\theta =_{ci} s_i$.

Lemma 4.1: Suppose in the definition of $M(P)$ only standard substitutions are considered, the refined **notion** of rule satisfaction is used, and each added fact, which is not with **predicate name prefix** *funnel_up_*, is standardized. Let $\overline{M}(P)$ be the resulting model. Then, $\overline{M}(P+funnel)/P =_{ci} M(P)$.

Proof: By Theorem 1, it suffices to show that $P+funnel$ derives a fact via refined satisfaction using rule r' during model construction iff

P derives the same fact, via pre-refined satisfaction, using rule r during model construction $M_1(P)$ (as defined in the statement of Theorem 1).

In forming $M_1(P)$ each added fact is standardized and w.l.o.g. only standard substitutions are used. In forming $\overline{M}(P)$ this is also the case except that ordinary matching is used with $funnel\_up$ literals and $funnel\_up$ facts are not standardized.

Thus, it suffices to show that for a standard fact I and standard substitution θ, $t_1\theta =_{ci} I$

iff

$funnel\_up\_t_1$ in r' can be matched via θ with fact $funnel\_up\_t_1\theta$ in forming the model $\overline{M}(P+funnel)$.

By Theorem 2, $t_1\theta =_{ci} I$ iff there exists a permutation $permute\_i\_t_1$ of t_1 such that $permute\_i\_t_1\theta =_i I$. By construction there is a rule $funnel\_up\_t_1 \leftarrow permute\_i\_t_1$ in $P+funnel$.

So, for a standard fact I and standard substitution θ, $t_1\theta =_{ci} I$

iff

the body of some permutation rule $funnel\_up\_t_1 \leftarrow permute\_i\_t_1$ i-matches I via θ

iff

In forming $\overline{M}(P+funnel)$, $funnel\_up\_t_1\theta$ is added

iff

$funnel\_up\_t_1$ in r' can be matched via θ with $funnel\_up\_t_1\theta$ in forming the model $\overline{M}(P+funnel)$. []

3.2. The second step

In the next step of the transformation, each permutation rule $funnel\_up\_t_1 \leftarrow permute\_i\_t_1$ is deleted and replaced with, usually many, *generic rules* obtained from $G(t)$, where $t = permute\_i\_t_1$. For each pair (g,ω) in $G(permute\_i\_t_1)$ the rule $funnel\_up\_t_1\omega \leftarrow g$ is added, g is called a *generic literal*.

Continuing the previous example, let us concentrate on one particular permutation rule, say $funnel\_up\_friends(set\_of(X,Y,john)) \leftarrow friends(set\_of(X,john,Y))$. For the simple *set_of* term in the body of this rule, each ω can be represented by indicating which arguments were identified as equal by ω. Once this is done, a bottom-to-top compaction obtains g. The possibilities can be represented symbolically as patterns (#,#,#), (#,@,#), (@,#,#), (#,#,@), (#,@,&). Each such possibility has implications on the values assigned to variables in the rule. The first possibility (#,#,#) implies that θ must assign *john* to both X and Y. Thus we generate a rule:

(a) funnel_up_friends(set_of(john,john,john)) ← friends(set_of(john)). (#, #, #)

The second possibility (#,@,#) implies that θ must assign the same values to X and Y. Thus we generate a rule:

(b) funnel_up_friends(set_of(X,X,john)) ← friends(set_of(X,john)). (#, @, #)

For the other possibilities we generate, respectively:

(c) funnel_up_friends(set_of(X,john,john)) ← friends(set_of(X,john)). (@, #, #)
(d) funnel_up_friends(set_of(john,Y,john)) ← friends(set_of(john,Y)). (#, #, @)
(e) funnel_up_friends(set_of(X,Y,john)) ← friends(set_of(X,john,Y)). (#, @, &)

After we do the above for each permutation rule we end up with a large collection of new generic rules and no permutation rules.

Define P*+generic* as the resulting program following the transformation. Refine rule satisfaction by adding: "(c) $t_i\theta = s_i$ if t_i is a generic literal," (the definition precedes Lemma 4.1).

Lemma 4.2: Suppose that in the definition of $M(P)$ only standard substitutions are considered, the newly refined notion of rule satisfaction is used, and each added fact, which is not with predicate name prefix *funnel_up_*, is standardized. Let $\hat{M}(P)$ be the resulting model. Then, $\hat{M}(P+generic)/P =_{ci} M(P)$.

Proof: By Lemma 4.1, it suffices to show that $\overline{M}(P+funnel)=\hat{M}(P+generic)$. By Theorem 3, if I is a standard fact, there exists a standard substitution θ such that $permute\_i\_t_1\theta =_i I$ iff there exists a substitution δ and an E-sequence E, inducing a satisfiable $Q(E)$ via mgu ω and a generic $g=E(permute\_i\_t_1)\omega$ such that $permute\_i\_t_1\omega =_i g$, $g\delta = I$ and $\omega\delta$ is standard.

By generic rule construction, a fact $funnel\_up\_t_1\theta$ is added by a permutation rule in forming $\overline{M}(P+funnel)$ using i-matching of $permute\_i\_t_1$ via standard θ to a standard fact I

iff

there is a generic rule that will add $funnel\_up\_t_1\omega\delta = funnel\_up\_t_1\theta$ in forming $\hat{M}(P+generic)$ using ordinary matching of g to I with θ,ω,δ and g as in Theorem 3.

It follows that any fact in $\overline{M}(P+funnel)$ is also in $\hat{M}(P+generic)$ and that every fact derived for $\overline{M}(P+generic)$ by a generic rule, induced by generic term g and mgu ω, via a δ such that $\omega\delta$ is standard is also in $\overline{M}(P+funnel)$. Thus, $\overline{M}(P+funnel)=\hat{M}(P+generic)$ []

Suppose a body g of a rule with head $funnel\_up\_t_1\omega$ matches a standard fact I with standard substitution δ. This produces a fact $funnel\_up\_t_1\omega\delta$. This fact is matched with $funnel\_up\_t_1$ via $\omega\delta$. By Claim 1(ii), ω is strongly compact, since δ is standard, so is $\omega\delta$ (otherwise, I being standard is contradicted). This implies that indeed following the rewriting, the (ordinary) matching of $funnel\_up\_t_1$ and a fact, is done via a standard substitution. Thus, in Lemma 4.2, there is no generality lost in considering only standard substitutions.

The transformation above is applied to a single literal in a single rule. Clearly, it can be applied to all literals in a rule which contain set_of subterms until they are all "converted" into funnel_up literals. Similarly, each program rule can be separately rewritten. (Of course, care must be taken to avoid naming conflicts; e.g. if t appears in rule r_1 and in rule $r2$ then we may use $funnel\_r1\_up\_t$ in rewritting r_1 and $funnel\_r2\_up\_t$ in rewritting $r2$.) Call the result *the transformed P*, denoted P'. Based on Lemma 4.2, and the observation following that Lemma, we conclude that if derived facts (other than those derived for generic rules) are standardized in computing $M(P')$, then $M(P')$ may be computed by only considering ordinary matching.

3.3. The third step

In the previous step each permutation rule was replaced with some generic rules. We now describe the next stage in the transformation which we call *body homogenizing*. Recall that terms s, t sharing no variables are variants if there exists a substitution θ which is a 1-1 renaming of variables such that $s\theta = t$. It may happen that in the collection of generic rules produced above, we may locate two rules, $r_1{:}head_1 \leftarrow body_1$ and $r_2{:}head_2 \leftarrow body_2$, such that $body_1$ and $body_2$ are variants. Since the meaning of a program is not altered when the variables in a rule are consistently renamed, we can rewrite r_1 as $head_1\theta \leftarrow body_2$ (since $body_1\theta = body_2$). Consequently, we can rewrite the collection of rules in such a way that all bodies which are variants of each other become now syntactically identical. As an illustration consider the pattern $(@,\#,\#)$ and the permutation rule with the body $friends(set\_of\,(john, Y, X))$. Note that this is a different permutation rule than the one we considered before, with body $friends(set\_of\,(X, john, Y))$, that induced rules (a)-(e). The rule that we get is:

(f) funnel_up_friends(set_of(X, X, john)) ← friends(set_of(john, X)).

The body of rule (d), $friends(set\_of\,(john, Y))$, is a variant of the body of rule (f) viz. $\theta = \{Y/X\}$. Thus, we rewrite rule (d) as:

(d') funnel_up_friends(set_of(john, X, john)) ← friends(set_of(john, X)).

Once rule-bodies are homogenized we can rewrite them in MHSB format (S stands for Single), by associating with each body all of the heads appearing in rules in conjunction with this body. Of course, if two heads grouped for a body are equal, only one is retained.

Example 4: The final result for our example are the following MHSB rules:

(1) funnel_up_friends(set_of(X,Y,john)),
funnel_up_friends(set_of(Y,X,john)) ← friends(set_of(X,john,Y)).

(2) funnel_up_friends(set_of(X,Y,john)),
funnel_up_friends(set_of(Y,X,john)) ← friends(set_of(X,Y,john)).

(3) funnel_up_friends(set_of(X,Y,john)),
funnel_up_friends(set_of(Y,X,john)) ← friends(set_of(john,X,Y)).

(4) funnel_up_friends(set_of(X,X,john)),
funnel_up_friends(set_of(john,X,john)),
funnel_up_friends(set_of(X,john,john)) ← friends(set_of(X,john)).

(5) funnel_up_friends(set_of(X,X,john)),
funnel_up_friends(set_of(john,X,john)),
funnel_up_friends(set_of(X,john,john)) ← friends(set_of(john,X)).

(6) funnel_up_friends(set_of(john,john,john)) ← friends(set_of(john)).
[]

3.4. Summary of the transformations on a rule

(1) replace the literal t in the original rule body with a *funnel_up_t* literal.

(2) For each permutation of t generate a permutation rule whose head is *funnel_up_t* and whose body is the permutation of t.

(3) Replace each permutation rule with a set of generic rules.

(4) Perform body homogenizing by making variant bodies syntactically identical.

(5) Group rules into MHSB format by associating with each body form all of the distinct heads it derives.

(6) Possible optimizations utilizing the rule body containing t ; see next section.

4. Optimization

The set of rules produced by the previous rewriting transformations offer significant opportunities for compile-time optimization. In this section, we discuss the elimination of rules that are redundant as result of (i) equalities and inequalities in the rules, (ii) storing the *set_of* terms in a standard sorted form and (iii) variables playing synonymous roles in rules.

4.1. Using equalities and inequalities

In some cases it may be determined that funnel-up heads in a MHSB rule cannot supply any bindings for which the whole (modified) rule body can be satisfied in matching all literals; in such cases these heads are disposed of in advance. Such cases often involve arithmetic predicates and the predicates = and ≠. For example, the head *funnel_up_friends(set_of(john, X, john))*, can be discarded from the MHSB rule (4) in Example 4, as it will force $X = john$ in the original rule and thus it violates $X \neq john$. Thus, rule (4) can be replaced by (4') below:

(4') funnel_up_friends(set_of(X,X,john)),
funnel_up_friends(set_of(X,john,john)) ← friends(set_of(X,john)).

At compile-time some violations can be checked for as follows. Rename variables so that each rule has a set of variables disjoint from the set of variables in any other rule. Unify *funnel_up_t* in the body of the modified rule r' with h, the head of the checked MHSB rule; let θ be the mgu that was used. Now consider an equality constraint $q = s$ in r'. If $q\theta$ and $s\theta$ are not ci-unifiable, then h can be discarded. Checking this can be done by using a ci-unification procedure; the description of such a procedure is outside the scope of this paper. Next consider an inequality constraint $q \neq s$ in r'. We consider it violated at compile-time only if $q\theta =_{ci} s\theta$ which can also easily be checked.

4.2. Using the standard representation assumption

In other cases it may be determined that a body of a MHSB rule will never match a standard fact. For example, if *friends(set_of(john, eric, X))* happens to be a body in a MHSB rule then it cannot match any standard fact because *eric* precedes *john* in the sorted order. A term is *mismatching* if it cannot match any standard fact I. The decision problem as to whether a given term is mismatching is still open. However, we make the following observations.

We say that a given term t is *antiordereded* if it contains a *set_of* subterm s such that for all substitutions θ such that $t\theta$ is ground, $s.j\theta$ precedes $s.i\theta$ in the total order on terms where $s.j$ $(s.i)$ is the j'th (i'th) argument of s, $i < j$. For instance, $f(g(1), set\_of(male(X), male(Y), female(Z)))$ is antiordered since *female* precedes *male*. Observe that a term may be mismatching and yet not be antiordered, e.g.,

$t = f(set\_of(1,X), set\_of(X,1))$, each *set_of* subterm of t by itself can match with a standard fact yet t cannot. We have the following:

Lemma 5.1: An antiordered term is mismatching. []

So, if a generic literal is antiordered, and hence mismatching, the generic rule for this generic literal will never be satisfied and therefore can be discarded.

We now present a method that detects many cases, but not all, in which a term t is antiordered. For term t, if t is a constant then $t[0]$ denotes t and otherwise $t[0]$ denotes the main functor of t. We need the following procedure, *precedes*, which defines a relation R on terms ($T_1 R\ T_2$ iff $precedes(T_1, T_2)$ returns **true**). In R, for all variables X, for all terms T, $X\ R\ T$ and $T\ R\ X$. When R is restricted to ground terms, it reduces to the total order on ground terms defined previously. Thus, one can think about the relation R as a "generalization" of the relation $\leq$ on ground terms.

```
procedure precedes (t,s):boolean ;
/* variables are magically ok, we "approximate" here */
if t or s is a variable then return true;
if t[0] precedes s[0] in the total order on terms then return true;
if t[0] follows s[0] in the total order on terms then return false;
if t[0]=s[0] and t[0] is a constant then return true;
if t[0]=s[0] then
 begin /* need to compare arguments if same functor */
  continue:=true;
  i:=1;
  while i ≤ arity(t) and i ≤ arity(s) and continue do
   begin
    if t[i]≠s[i] then
     /* determine if t[i] precedes s[i] and exit loop */
      begin
       continue:=false;
       if precedes(t[i],s[i]) then comp:=true else comp:=false
      end;
    i:=i+1; /* compare next arguments in t and s */
   end;
  /* check if loop exited with all checked pairs equal, i.e. continue=true */
  if continue then comp := arity(t) ≤ arity(s);
  return comp
 end;
```

We state without proof that if $precedes(t,s)$ returns **false** then for all substitutions θ, $s\theta$ precedes $t\theta$. Thus, to determine whether t is antiordered we can use the following method. Apply *precedes* to each pair of arguments at positions i, j, $i < j$, in each *set_of* subterm of t. If any such application returns **false** then t is antiordered.

We now consider the computational complexity of detecting antiordered terms using the above method. First, in *precedes* the line "**if** $t[i] \neq s[i]$ **then**" takes time O(size of $s[i]$ + size of $t[i]$). So, *precedes*(s, t) is O((size of t+size of s)^2). Second, given t we need to apply *precedes* to each pair of arguments in a *set_of* subterm of t. The number of such pairs is O((size of t)^2). Thus our method is O((size of t)^4). The 4 in the exponent can easily be reduced to 3 by locating the first point of "disagreement" in checking "**if** $t[i] \neq s[i]$ and calling *precedes* recursively on the corresponding subterms.

More stringent criteria could also be considered to enhance *precedes*. One possibility is that the ordering determined through the execution between variables and terms can in simple cases be checked for acyclicity. For instance, on *set_of(X, Y, f(Y), f(X))* procedure *precedes* returns **true**. Observe that no matching is possible since, once X and Y are instantiated, we cannot have both X precedes Y and Y precedes X in the total order on terms. However, procedure precedes is computationally feasible and detects many cases in which t is antiordered.

4.3. Using Synonyms

Other optimization techniques are similar to tableaux minimization [ASU79]. A *distinguished substitution* w.r.t. t is a substitution θ which assigns to each variable X appearing in t a unique distinct constant which does not appear in t or in the program P. For our purposes this substitution is unique, assigning unique constant x to variable X. The *distinguished binding form* of t, t_b, is obtained by applying to t the distinguished substitution w.r.t. t. An *expression* is a term, a literal or a rule. Given a set of expressions S, a binding θ is *reducing* w.r.t. S if it transforms each element of S into its distinguished binding form, i.e., converting S into a set of ground terms in which S's variables are uniformly renamed into distinct constants.

Rule bodies $body1 = B_1, \ldots, B_n$ and $body2 = C_1, \ldots, C_n$ are *isomorphic*, denoted $body1 =:= body2$ if $set\_of\,(B_1, \ldots, B_n) =_{ci} set\_of\,(C_1, \ldots, C_n)$. Here, we represent $s = t$ as $=(set\_of\,(s,t))$ and we represent $s \neq t$ as $\neq(set\_of\,(s,t))$. Consider a funnel-up heads h_1 and h_2 in a MHSB rule m for literal t. let P' be the result of the rewriting of P. Funnel-up heads h_1, h_2 in m are *synonyms* if deleting from m in P' either the head h_1 or the head h_2, results in an equivalent program $\bar{P}$, i.e. one that generates correct results for queries, as asked against P. We define the following *synonym test*. Let h_{ib} be the distinguished binding version of h_i , $i = 1,2$ produced by reducing binding β w.r.t. h_1 and h_2. For $i = 1,2$, suppose that θ_i matches $funnel\_up\_t$ in r' with h_{ib} (which is the distinguished binding form of h_i). Let

$$r_i' = (r - t)\theta_i = head_i' \leftarrow body_i'$$

where $(r - t)$ is r after deleting the literal t from its body. Then, the synonym test succeeds if $body_1' =:= body_2'$ and $head_1' =_{ci} head_2'$.

We illustrate the above definitions via the following example; consider the rule r:

john_friend(X) ← friends(set_of(X,Y,john)), X $\neq$ john, nice(X).

Here $t = funnel\_up\_friends\,(set\_of\,(X,Y,john))$. We now examine a MHSB rule, for example rule (4') discussed above:

(4') funnel_up_friends(set_of(X,X,john)),

funnel_up_friends(set_of(X,john,john)) ← friends(set_of(X,john)).

After applying the distinguished substitution $\alpha = \{X/x\}$ to the two heads in rule (4') we obtain:

h_{1b} = funnel_up_friends(set_of(x,x,john))

h_{2b} = funnel_up_friends(set_of(x,john,john)).

Thus,

$\theta_1 = \{X/x, Y/x\}$ and $\theta_2 = \{X/x, Y/john\}$.

Also,

$(r-t)=$ john_friend(X) $\leftarrow$ X $\neq$ john, nice(X).

So,

$r_1'=$ (john_friend(X) $\leftarrow$ X $\neq$ john, nice(X))$\theta_1=$ (john_friend(x) $\leftarrow$ x $\neq$ john, nice(x)), where

$head_1'=$ john_friend(x), and $body_1'=$ set_of(x $\neq$ john, nice(x)); and

$r_2'=$ (john_friend(X) $\leftarrow$ X $\neq$ john, nice(X))$\theta_2=$ (john_friend(x) $\leftarrow$ x $\neq$ john, nice(x)), where

$head_2'=$ john_friend(x), and $body_2'=$ set_of(x $\neq$ john, nice(x)).

Consequently, $head_1'=head_2'$ and $body_1'=body_2'$. Since $=$ implies $=_{ci}$, the synonym test succeeds on *funnel_up_friends*(*set_of* (*X*,*X*,*john*)) and *funnel_up_friends*(*set_of* (*X*,*john*,*john*)). Based on Theorem 4 below, they are synonyms and either may be eliminated, for instance, the latter. Similar optimization steps can be applied to rule (5) of Example 4, thus yielding the rules of Example 3.

The correctness of the synonym test follows from the following Theorem.

Theorem 4: If the synonym test applied to h_1 and h_2 succeeds, then h_1 and h_2 are synonyms.

Proof: Since $body'\,\theta_1 =:= body'\,\theta_2$, substituting for each variable in the MHSB rule m any ground term, as expressed by a substitution α, yields

$$body'\,\theta_1\beta^{-1}\alpha =:= body'\,\theta_2\beta^{-1}\alpha$$

and

$$head'\,\theta_1\beta^{-1}\alpha =_{ci} head'\,\theta_2\beta^{-1}\alpha$$

where β^{-1} is the inverse of the grounding substitution producing $h_{i\,b}$, $i=1,2$.

The following observation can be proven by induction: if $set\_of\,(t_1,\ldots,t_n) =_{ci} set\_of\,(s_1,\ldots,s_n)$ then for all $1 \le i \le n$ there exists $1 \le j \le n$ such that $t_i =_{ci} s_j$. Suppose we are given a set of standard facts M_i.

Consider the construction of $\hat{M}(P+generic)$. It follows from the above observation that tuple *standard* (***head'*** α) is added during model construction using r' by matching, using α, $funnel\_up\_t$ with a tuple generated by $head_1$ and ci-matching each literal in $(r-t)$ with a fact in M_i,

iff

an equal tuple, i.e. *standard* (*head'* α), is added during model construction by r' by matching, using α, $funnel\_up\_t$ with a tuple generated by $head_2$ and ci-matching each literal in $(r-t)$ with a fact in M_i.

Hence, deleting *head* (which is either h_1 or h_2) results in a program $P+generic-head$ such that $\hat{M}(P+generic)/P =_{ci} \hat{M}(P+generic-head)/P$. Therefore, h_1 and h_2 are synonyms. []

The above implies that if the synonym test succeeds on h_1,h_2 then only one of h_1,h_2 need be retained in m. An obvious optimization procedure is to repeatedly test for synonyms and remove heads accordingly.

4.4. Additional Optimization Techniques

The multihead rule representation creates an opportunity for further optimizations– which, for the most part have been implemented in the LDL system. One such optimization is a generalization of the synonym test, which often entails further rule elimination. This is briefly discussed in the Appendix.

A second optimization opportunity arises at code generation time, when multi-head rules having the same body can be grouped into multi-head multi-body rules (such as a,b ← c,d ; e,f ; g,h described in the introduction). which can then be compiled as units. Then, as different bodies are considered for possible matchings, it is possible to take into account the results of matching the previous bodies to eliminate unnecessary work. This is done through a Prolog-like backtracking mechanism which always uses as much information as possible each time a new matching is tried out. The same general idea applies to generated tuples in the multihead part. These tuples introduce certain variations of each other, thus the "next" tuple to be generated may be obtained by a minor permutation on a previously generated one. By examining the heads an "inexpensive" sequence may be obtained. Furthermore, some variables in t' are used in r' *only* in t'. Intuitively, such variables check "existence". The terms in corresponding positions in funnel-up heads *need not be formed at all.*.

5. Conclusions

The approach presented for supporting sets in a HCLPL represents a clear advancement of the state of the art. First of all, it eliminates the need to use E-matching at run time in supporting sets; instead we compile the original program into one that only requires ordinary matching. Second, it leads to more efficient implementations since the rewritten program is optimized using information available in the given rule, thus eliminating many of the alleys explored by the blind search of E-matching. In particular we take advantage of having a standard representation for facts, of inequality constraints and synonyms (i.e. matchings that lead to the generation of identical tuples).

Some of the techniques, e.g. multiheads, are still in the experimental stage and we expect to further report on them in the future. Other aspects are now being explored, among these are the support for the standard set operations, e.g. member, equality, inequality, union. The problem of whether given a term t, t is mismatching, i.e. cannot match with any standard fact, is still open.

We should note that the rewriting is expensive and may take exponential time in the size of the rewritten term. Thus, for sets with more than ten items or so it is not very practical, as it may result in too many rules; this is not surprising in light of [KN86]. For large sets we can resort to using other techniques which use the predicate member. (These techniques are outside the scope of this paper.) For "small" sets these other techniques are not as efficient as the methods described in this paper. Also, in many cases of such large sets, many of the *set_of* arguments are variables that appear in one place and nowhere else in the rule, these are "placeholders" used to indicate cardinality. It will be interesting to "grow" the rewritten rule from a version produced by first ignoring these "place-holders" and then adding them one at a time.

Acknowledgments

We would like to thank C. Beeri for reading and commenting on an early version of this paper. We would like to thank Y. Sagiv for his many useful comments. We also would like to acknowledge the help of Nissim Francez, Roger Nasr and Jim Christian while revising the paper. Lastly, the referees did an excellent job in supplying us with relevant references which resulted in better understanding of the problem we address and and its relationship to results in the literature.

References

[AB87] Abiteboul, S. and C. Beeri. *On the Power of Languages for the Manipulation of Complex Terms*. Unpublished Manuscript.

[ASU79] Aho, A.V., Y. Sagiv and J.D. Ullman, *Equivalence of Relational Expressions*. SIAM J. Computing Vol. 8, No. 2, 218-246, 1976.

[BHKSST88] Buckert, H. J., A. Herold, D. Kapur, J. H. Siekmann, M. H. Stickel, and M. Tepp, *Opening the AC-Unification Race*, J. Automated Reasoning, 4, 465-474, 1988.

[BNRST87] Beeri, C., S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. *Sets and negation in a Logic Database Language (LDL1)*, 6th ACM Symposium on Principles of Database Systems, San Diego, CA., 1987.

[BUTT86] Buttner, W., *Unification in the Data Structure Sets*, Proc. 8th International Conference on Automated Deduction (CADE-8) Oxford, England, (also Springer Verlag LNCS-230,) July 1986.

[CHRIS89] Christian, J., *High-Performance Permutative Completion*, Dissertation, Univ. of Texas, Austin, MCC Tech. Rep. ACT-AI-303-89, 1989.

[CM84] Clocksin, W.F., and C.S. Mellish. *Programming in Prolog*, 2n d. Edition, Springer Verlag, 1984.

[DERSH87] Dershowitz, N., *Completion and its Applications*, Coll. on the Resolution of Equations in Algebraic Structures, Austin, Texas, May 1987.

[FAY79] Fay, M., *First Order Unification in an Equational Theory*, Proc. 4th International Conference on Automated Deduction (CADE-4), Austin, Texas, 161-167, 1979.

[FAGES87] Fages F., *Associative-commutative Unification*. J. Symbolic Comp., 3-3, 257-275, June 1987.

[HERO86] Herold, A., *Combination of Unification Algorithms*, Proc. 8th International Conference on Automated Deduction (CADE-8) Oxford, England, (also Springer Verlag LNCS-230, 450-469,) July 1986.

[HUET80] Huet G., *Confluent Reductions: Abstract Properties and Applications to term rewriting systems*. J. ACM, Vol 27, No. 4, 797-821, October 1980.

[HULL80] Hullot J-M, *Associative Commutative Pattern Matching*, Proc. 5th International Conference on Automated Deduction (CADE-5), Springer-Verlag LNCS 87, 318-334, 1980.

[JK86] Jouannaud J-P., and H. Kirchner, *Completion of a Set of Rules Modulo a Set of Equations*, SIAM J. COMPUT., Vol. 15, No. 4, 1155-1178, November 1986.

[JKK83] Jouannaud J-P., C. Kirchner and H. Kirchner, *Incremental Construction of Unification Algorithms in Equational Theories*. ICALP '83, (also Springer Verlag LNCS-154, 361-373,) 1983.

[KN86] Kapur, D., and P. Narendran. *NP-Completeness of the Set Unification and Matching Problem*, Proc. 8th International Conference on Automated Deduction (CADE-8) Oxford, England, (also Springer Verlag LNCS-230,) July 1986.

[KRS84] Korth, H.F., M.A. Roth, and A. Silberschatz, *A Theory of Non-First-Normal-Form Relational Databases*, 1984.

[KV84] Kuper, G.M., and M.Y. Vardi. *A new Approach to Database Logic*, Proc. Third ACM Symp. on Principles of Database Systems, Waterloo, Canada, 1984.

[LLOY87] Lloyd, J.W., *Foundations of Logic Programming*. Springer Verlag, 1987.

[LC88] Lincoln, C., and J. Christian, *Adventures in Associative-Commutative Unification (a summary)*, Proc. 9th International Conference on Automated Deduction (CADE-9), (also in Springer Verlag LCNS 310, 358-367,) 1988.

[LS76] Livesey, M., and J. H. Siekmann, *Unification of A+C-Terms (Bags) and A+C+I-Terms (Sets)*. Universitat Karlsruhe, Fakultat fur Informatik Technical Report 5/76, 1976.

[MK85] Maluszynski J. and H. J. Komorowski, *Unification-Free Execution of Logic Programs*, Proc. 1985 Symposium on Logic Programming, 78-87, IEEE Computer Society Press, 1985.

[NATS89] Naqvi, S. A., S. Tsur, *A Logical Language for Data and Knowledge Bases* W. H. Freeman, 1989.

[OO83] Oszoyoglu, G. and Z. Oszoyoglu. *An Extension of Relational Algebra for Summary Tables*, Proc. 2nd. International (LBL) Conference on Statistical Database Management, 1983.

[PS81] Peterson, G. E. and M. E. Stickel. *Complete Sets of Reductions for Some Equational Theories* , J. ACM, Vol. 28, No. 2, 233-264, April 1983.

[RETY87] Rety, P., *Improving Basic Narrowing Techniques*. Conf. on Rewriting Techniques and Applications, Bordeaux, May 1987.

[RSSU79] Raulefs, A. P., J. H. Siekmann, P. Szabo, and E. Unvericht, *A Short Survey on the State of the Art in Matching and Unification Problems*, ACM Sigsam Bulletin 13,2, 14-20, May 1979.

[SIEK89] Siekmann J. H., *Unification Theory*, J. Symbolic Computation, (1989) 7, 207-274.

[STICK81] Stickel, M.E., *A Unification Algorithm for Associative-Commutative Functions*, J. ACM 28,3, 423-434, July 1981.

[SZ87] Sacca, D., and C. Zaniolo, *Implementation of Recursive queries for a Data Language Based on Pure Horn Clauses*, Fourth International Conference on Logic Programming, Melbourne, Australia, 1987.

[TZ86] Tsur, S., and C. Zaniolo *LDL: A Logic-Based Data-Language*, Proc. 12th Int Conf. on very Large Databases, Kyoto, Japan, 1986.

[ZAN88] Zaniolo, C., *Design and Implementation of a Logic Based Language for Data Intensive Applications*, Proc. of the Fifth International Conference and Symposium on Logic Programming, 1667-1687.

Appendix

In many cases more than a single conclusion, i.e. head tuple, may be drawn from a single match of the body literals with facts. Notationally, we indicate this by rewriting the rule in a MHSB format.

Example 5: Consider

r: john_friend(X) ← friends(set_of(X,Y,john)),nice(X),nice(Y)

Its transformed version according to the previous section is

r': john_friend(X) ← funnel_up_friends(set_of(X,Y,john)), nice(X), nice(Y)

Suppose the body is matched with facts *friends(set_of (al ,jim ,john)),nice(al)* and *nice(jim)*. The deduced head tuple is *john_friend(al)*. Intuitively, as *al* and *jim* play a totally symmetric role, *john_friend(jim)* may be deduced as well. Hence, the rule is rewritten as $\bar{r}$:

$\bar{r}'$: john_friend(X), john_friend(Y) ← funnel_up_friends(set_of(X,Y,john)), nice(X), nice(Y). []

The main advantage of identifying multiheads for a rule is that it enables further eliminations of funnel-up heads.

Example 6: Consider a MHSB rule *m* generated for Example 5, for the generic literal friends(set_of(john,X)):

funnel_up_friends(set_of(john,X,john)),
funnel_up_friends(set_of(X,john,john)),
funnel_up_friends(set_of(X,X,john)) ← friends(set_of(john,X)).

If the original rule is kept as is, i.e. r', then the three heads in *m* must be retained. However, if the rule is modified to the form $\bar{r}'$ then one of the heads in *m* may be eliminated, resulting in:

funnel_up_friends(set_of(X,john,john)),
funnel_up_friends(set_of(X,X,john)) ← friends(set_of(john,X)). []

The deletion of heads in *m* implies that fewer matchings are performed in the body of $\bar{r}'$ with facts generated by funnel-up heads as compared to the matchings performed in r'. This saves on checking for matchings in the rest of the body literals in $\bar{r}'$. We should note that in some cases the above transformation may result in a slight cost increase.

Example 7: Consider the MHSB rule *w* for the generic literal friends(set_of(john)):

funnel_up_friends(set_of(john,john,john)) ← friends(set_of(john)).

Here, for a single match with this rule *w*, $\bar{r}'$ will, wastefully, produce two identical heads of the form *john_friend(john)*. []

This apparent waste is marginal as it involves simple value permutations at run-time to produce deduced tuples for the multiple heads in $\bar{r}'$ as opposed to matching with possibly numerous tuples.

The first problem in forming a rule like $\bar{r}$ is how to obtain additional head tuples based on a *single* binding to body variables. Some additional notation is needed. A *variable to variable mapping (vvmap)* is a substitution $\{X_1/Y_1, \ldots, X_n/Y_n\}$ where $X_1, \ldots, X_n$ are distinct variables and $\{X_1, \ldots, X_n\}=\{Y_1, \ldots, Y_n\}$. Let E be an expression and θ a vvmap, θ is *preserving* with respect to E if $E\theta =_{ci} E$. For example, if $E=set\_of\,(q(X,Y), q(Y,X), p(set\_of\,(X,Y,Z)))$ then $\theta=\{X/Y, Y/X\}$ is preserving while $\theta=\{X/Z, Z/X\}$ is not preserving. If r is a rule, with body $B_1, \ldots, B_n$, then θ is a vvmap (respectively, preserving vvmap) w.r.t. r if θ is a vvmap (respectively, preserving vvmap) w.r.t. $set\_of\ (B_1, \ldots, B_n)$.

We would like to obtain all solutions derivable from a body under all different preserving vvmaps. This is because of the following key observation:

Observation A.1: Let θ be a preserving vvmap w.r.t. $head \leftarrow body$. For any matching α of *body* with standard facts deriving head tuple $head\,\alpha$, there is another matching, *with the same standard facts,* such that the head tuple $head\,\theta\alpha$ is derived.

Proof: Let $body = B_1, \ldots, B_n$. Consider standard facts $d_1, \ldots, d_n$ and a matching α such that for $i = 1, \ldots, n$, $B_i \alpha =_{ci} d_i$. Each B_i is mapped by θ to $B_i\theta$; since θ is a preserving vvmap, there exists some B_j, $1 \leq j \leq n$ such that $B_i\theta =_{ci} B_j$. Denote the smallest such j as $\theta(i)$. Now, we can match $B_1, \ldots, B_n$ to $d_1, \ldots, d_n$ in a different way, namely, B_i is matched with $d_{\theta(i)}$, by matching each variable X in B_i with what $X\theta$ in $B_{\theta(i)}$ was matched with in $d_{\theta(i)}$. Thus whenever $head\,\alpha$ can be produced, so can $head\,\theta\alpha$. []

We can extend the definition of $M(P)$ (respectively, $\overline{M}(P)$, $\hat{M}(P)$) to the case where original rules are in MHSB format, simply by stating that $h\,\theta$ (respectively, $standard(h\,\theta)$) are added during model forming for all heads h in rule $\bar{r}$. We use $\bar{r}'$ to denote $\bar{r}$ once t is replaced with $funnel\_up\_t$ in the transformation.

Corollary: If θ is a preserving vvmap for rule r: $head \leq\!\!-\!\!-body$, then replacing in P r with $\bar{r}$ results in the same $M(P)$ (respectively, $\overline{M}(P), M(P)$ for $\bar{r}'_i$), where $\bar{r}$: $head\,,\ head\,\theta \leftarrow body$. []

Thus, to each original rule body we may attach many heads, one per each preserving vvmap θ. Clearly, this results in an equivalent program. Of course, if a number of heads thus generated are ci-equal, only one need be retained.

The redundancy elimination implied by Theorem 4, may be easily adapted to the situation where original rules are transformed into MHSB equivalent representation. Head $head_1$ in m is *dominated* if deleting $head_1$ results in an equivalent program.

We now define a *domination test* to take into account the fact that $\bar{r}$ is MH. Intuitively, $head_1$ is dominated because of $head_2$ if, for the generic literal match in m's body, the multiheads after unifying with a $head_2$ generated tuple, form a superset, modulo commutativity and idempotence, of the multiheads after unifying with a $head_1$ generated tuple. Define $S \subseteq * \subseteq S'$ if both S and S' are sets and for each $A \in S$ there exists $B \in S'$ such that $A =_{ci} B$.

The domination test, on funnel-up heads h_1, h_2 is as follows. Let $\bar{r}$ be a MH rule with set of heads H and body $body$. Let t' be a literal in $\bar{r}'$. Let h_{ib} be the distinguished binding version of $h_i, i = 1,2$. For $i = 1,2$, let θ_i match h_{ib} with t' in $\bar{r}$. Let $\bar{r}_i = (\bar{r} - t)\theta_i = \overline{H}_i \leftarrow \overline{body}_i$, $i = 1,2$, where $(\bar{r} - t)$ is obtained from $\bar{r}$ by deleting literal t. Then, the domination test determines that h_2 dominates h_1 if $\overline{body}_1 =:= \overline{body}_2$ and $\overline{H}_1 \subseteq * \subseteq \overline{H}_2$.

The domination test is in fact a generalization of the synonym test of the previous section, specializing it to the case where original rules may have a number of heads. While synonym is a symmetric relation, dominated is a one place relation. In a way similar to that in Theorem 4, it can be shown that when the domination test determines that h_2 dominates h_1, where both h_1 and h_2 are heads in a MHSB rule m, then h_1 is dominated in m and thus may be deleted without altering the meaning of the program.

It might be possible to remove additional m heads. Intuitively, the idea is that the heads produced in $\bar{r}'$ due to some head in m are, collectively, also produced by those heads in m that give rise to an isomorphic body when unified with t'.