REWRITING OF RULES CONTAINING SET TERMS

IN A LOGIC DATA LANGUAGE (LDL)

Oded Shmueli*, Shalom Tsur, Carlo Zaniolo

MCC, Austin, Texas

ABSTRACT

We propose compilation methods for supporting set terms in Horn clause programs, without using general-purpose set matching algorithms, which tend to run in times exponential in the size of the participating sets Instead, we take the approach of formulating specialized computation plans that, by taking advantage of information available in the given rules, limit the number of alternatives explored Our strategy is to employ *compile time* rewriting techniques and 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. Overview

We propose compilation methods for supporting set terms in Horn clause programs, without using general-purpose set matching algorithms Instead we take the approach of formulating specialized computation plans that, by taking advantage of information available in the given rules, limit the number of alternatives explored Our strategy is to employ rewriting techniques ***at compile time*** to transform the problem into an "ordinary" Horn clause compilation problem The execution cost of the 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

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 Set-objects are internally represented as terms whose main functor is *set_of* 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

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

Assume that the database[1] contains the following facts

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 []

While this paper deals with E-unification [FAGES87, RS79, STICK81, LS76, LC87] our stated goal here, which emphasizes compile-time transformations motivated by a large fact database, sets it

* Current address Department of Computer Science, Technion, Haifa, Israel 32000

[1] 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)

apart from most research in this area. Moreover, we do not assume associativity, since we are interested in arbitrarily deep nesting and thus assume that, say, {a,{b}} is different from {a, b} It should be noted that deep nesting can be handled in the context of associative-commutative unification by introducing extraneous functions that are neither associative nor commutative; e g {a,f({b})}

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 [LLOY84] For the purpose of this paper one can safely think of LDL as a pure HCLPL (with a distinguished functor - *set_of*) whose semantics is given by applying the immediate consequence operator T_P [LLOY84] until fixpoint – i e a "bottom-up" repeated "firing" The only difference between our T_P and the one in [LLOY84] is that instead of matching we use ci-matching as defined below

The *set_of* functors are 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 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

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

The third is $\gamma=$ {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 $\alpha=$ { 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 which is instantiated into a set containing (mathematically) one element Again, no fact is derived because of X$\neq$john

The following is an example of the usefulness of i-matching Suppose a team needs up to three persons The predicate *team* locates two member teams such that the two have been a team before (recorded in *old_team*) and the two members have the capabilities of an engineer, a scientist and a medical doctor

team(set_of(X,Y)) ←
ok_team(set_of(X,Y)), $X \neq Y$
ok_team(set_of(X,Y,Z)) ←
old_team(set_of(X,Y,Z)), engineer(X), scientist(Y), medical_doctor(Z)

The semantics of a program P with set terms is defined using ci-matching Thus, two programs are equivalent when they produce the same set of answer tuples modulo ci-equality Then, the compilation of P transforms it into an equivalent program that employs only ordinary matching Thus, the *set_of* terms in the transformed program can be 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[2],

[2] 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

and use a short hand notation called multi-head-multi-body (MHMB) rules In a MHMB rule, comma is to be read as "and", and a semicolon as "or" So, a rule with m bodies and n heads represents $m \times n$ ordinary rules, one for each body-head combination

Example 3: Consider rule r, The rewritten rule is r'

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

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

funnel_up_friends (set_of (X ,X ,john)),

funnel_up_friends (set_of (X ,john ,john))

and, *funnel_up_friends (set_of (john ,X ,john))*

need to be transmitted 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

The transformation result may seem bulky However as a result, run-time ci-matching on a per tuple basis is replaced 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

---

the original term in the body of a rule

There are eight sections Section 2 discusses technical aspects of augmenting a HCLPL with the *set_of* predicate 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 4 Optimization techniques are discussed in section 5 The merits of a preliminary rewriting, in which original program rules are made "multihead" is discussed in section 6 Section 7 presents some elementary observations concerning compilation of the rewritten program Section 8 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 $f(a_1, \ \ , a_n)$ where f is an n-ary function symbol and, for $i=1, \ \ , n$, a_i is a term which is called the ***argument of t of index i*** f is an n-ary function symbol unless $f = set\_of$ which is a distinguished function symbol that does not have a fixed arity A term is ***ground*** if it contains no variables A term t defined according to (i) or (ii) will be called ***simple***, and ***complex*** otherwise

A rule is a formula of the form

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

where A and each B_i, $1 \leq i \leq n$, are ***literals*** (or ***predicates***), 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 loosely use the the word "term" to refer to both actual terms or literals, since literals, syntactically, have the same form as terms

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

$Y_j \in \{X_1, \cdot\ ,X_m\}$.

A substitution $\theta=\{X_1/t_1, \ . \ , X_n/t_n\}$ where $t_1, \quad , t_n$ are all ***ground***, i e, contain no variables, is called a ***binding*** A term t_1 is said to be ***more general than (or a generalization of)*** 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 the most general unifier of two terms [LLOY84], it follows that

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

The most general solution for U will be called the ***most general unifier*** (mgu) for U It also follows that the most general solution for U is unique modulo variable renaming 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

Let *set_of* be a distinguished function symbol It is intended to model 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 $a_1, \ . \ , a_n$ (not necessarily distinct) These intuitive notions are captured formally as follows.

Term t ***derives*** term s ***modulo idempotence***, denoted $t \Longrightarrow_i s$, if either (i) $t=s$ or (ii) s is t with the exception that a subterm t_1 of t, $t_1=set\_of\ (x_1, \ldots, x_i, .., x_{j-1}, x_j, x_{j+1}, .., x_n)$, such that $x_i=x_j$, is modified by deleting x_j to obtain $s_1=set\_of\ (x_1, \quad, x_i, \quad ., x_{j-1}, x_{j+1}, \cdot \quad, x_n)$ in s s is obtained from t by an ***elementary compaction step from*** t ***to*** s Observe that $t \Longrightarrow_i s$ does not imply $s \Longrightarrow_i t$.

Term t ***derives*** term s ***modulo commutativity***, denoted $t \Longrightarrow_c s$, if either $t=s$, or s is t with the exception that a subterm t_1 of t, $t_1=set\_of\ (x_1, \quad, x_i, \quad, x_{j-1}, x_j, x_{j+1}, \quad, x_n)$, is modified by exchanging arguments x_i and x_j to obtain $s_1=set\_of\ (x_1, \quad, x_j, \quad, x_{j-1}, x_i, x_{j+1}, \ , x_n)$ in s s is obtained from t by a ***permutation step from*** t ***to*** s Observe that $t \Longrightarrow_c s$ iff $s \Longrightarrow_c t$

Term t ***derives term*** s ***modulo commutativity and idempotence***, denoted $t \Longrightarrow_{ci} s$, if either (i) $t \Longrightarrow_i s$, or (ii) $t \Longrightarrow_c s$ Let $\overset{*}{\Longrightarrow}_i$, $\overset{*}{\Longrightarrow}_c$ and $\overset{*}{\Longrightarrow}_{ci}$ be the transitive closure of $\Longrightarrow_i$, $\Longrightarrow_c$ and $\Longrightarrow_{ci}$, respectively If $t \overset{*}{\Longrightarrow}_i s$ then s is obtained from t by ***elementary compaction*** from t to s, if $t \overset{*}{\Longrightarrow}_c s$ then s is obtained from t by ***permutation from*** t ***to*** s Let $=_i$, $=_c$ and $=_{ci}$ be the symmetric and transitive closure of $\Longrightarrow_i$, $\Longrightarrow_c$ and $\Longrightarrow_{ci}$, respectively

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 ***unification*** is replaced by ***matching,*** 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 $t \Longrightarrow_i s$ 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

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

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

Given a term t, the ***compact form of*** t denoted $com(t)$, is a compact term obtained from t $com(t)$ can be computed by repeating the following until there are no more changes

> Find a *set_of* subterm s of t such that all of s's arguments are compact and s has two identical arguments, delete the argument with the highest index

It can be shown that $com(t)$ is unique. Clearly, $t \stackrel{*}{==>}_s com(t)$ and the sequence of elementary compaction steps is such that a subterm A is handled, i.e. being made compact, **only after all of its arguments have been handled and are compact.** Such an elementary compaction step is called a ***bottom-up compaction step*** and a sequence of bottom-up compaction steps is called a ***bottom-up compaction***.

Given a term t, a ***strong compact form of t*** is a strongly compact term obtained from t as follows (it is not unique in general):

> Consider $S=\{s \mid s=_c t\}$. It can be shown that S is finite. If all $s \in S$ are compact then t is strongly compact. Otherwise, if $s \in S$ is a non-compact term, then let $t=com(s)$ and repeat this step.

It can be shown that if t_1 and t_2 are strong compact forms of t then $t_1 =_c t_2$.

The following Lemma states that if I is strongly compact then $t \stackrel{*}{==>}_s I$ implies that there is a sequence of standard compaction steps leading from t to I. Intuitively, duplicates are being thrown from subterms of t in such a way that a *set_of* subterm is considered for duplicate elimination only after all of its *set_of* subterms have been considered. We need a technical definition. 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 $1+\max\{height(t_1), \ldots, height(t_n)\}$.

Lemma 2.1:[3] If I is strongly compact and $t \stackrel{*}{==>}_s I$ then I can be obtained from t via a standard compaction.

The following Lemma states that if $t =_s I$ and I is strongly 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 strongly compact.

Lemma 2.2: Let I be strongly compact. $t =_s I$ iff $t \stackrel{*}{==>}_s I$.

Next, we show that if I is strongly compact and $t =_{cs} I$ 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.

[3]Because of space limitations, all of the proofs of the Lemmas and Theorems stated in this paper have been omitted. A full version of this paper which includes the proofs appears in [STZ87].

Lemma 2.3: Let I be strongly compact. $t =_{cs} I$ iff there exists w such that $t \stackrel{*}{==>}_c w \stackrel{*}{==>}_s I$.

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., ASCII order).

(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 they are equal on all positions up to some position i for which 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 order defined above on facts.

We make the following 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 are ***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.

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 subterms is handled only after all its *set_of* subterms have been handled. It can be shown that *standard*(t) is unique and that $t \stackrel{*}{==>}_{cs} standard(t)$ which implies $t =_{cs} standard(t)$.

To illustrate the importance of the standard representation assumption, let us assume that, by contradiction, we admit in the database the pair of facts *p(set_of(1,2))* and *q(set_of(2,1))* which violates 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 as defined above.

2.4. Semantics

The semantics of LDL 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 $\neq$ of arity two which are written in infix notation. For simplicity, we view the database as part of the program. Substitution θ

satisfies the body of a rule $h \leftarrow t_1, \ \ \ , t_n$ in a set of facts S, if for $i=1, \ \ \ , n$, either (i) t_i has form $s_1 = s_2$ and $s_1\theta =_{ci} s_2\theta$, or (ii) t_i has form $s_1 \neq s_2$ and $s_1\theta \neq_{ci} s_2\theta$, or (iii) there exists $s_i \in S$ such that $t_i\,\theta =_{ci} s_i$

Definition of $M(P)$

The *model* of a program P, denoted $M(P)$ is defined thus Let $M_0 = \emptyset$ For $i > 0$,

$M_i = M_{i-1} \cup \{ h\,\theta \mid$ binding θ satisfies the body of a rule $r \in P$ in M_{i-1}, with h the head of $r \}$

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

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

$M_i' = M_{i-1}' \cup$

$\{ h\,\theta \mid$ standard binding θ satisfies the body of a rule $r \in P$ in M_{i-1}', with h the body of $r \}$

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

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

$M_i'' = M_{i-1}'' \cup$

$\{standard(h\,\theta) \mid$ standard binding θ satisfies the body of a rule $r \in P$ in M_{i-1}'', with h the head of $r \}$

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

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

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

It can be shown that if $M(P)$ above is replaced with S such that $S =_{ci} M(P)$ 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 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, the Theorem 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 $M(P)$

3. The Decomposition Theorems

3.1. 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 substitution and permutation

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 $t_1\theta =_i I$

3.2. 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 is tantamount to considering each possible standard compaction and solving a set of (ordinary) unification equations implied by the standard 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 an address 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 A is the term address in t whose content is the subterm $f(t_1, \ \ \ , t_n)$ then $A\,j$, $1 \leq j \leq n$, is a term address in t whose content is t_j

We use $t\,A$ to denote the subterm of t whose address is A (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 $A\,i = A\,j$ where A is the address of a *set_of* subterm of t, $i < j$ and $A\,i$ and $A\,j$ are addresses of arguments of $t A$ 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 standard 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 standard 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 standard 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 standard 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 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 $E = E_1, \quad , E_n$, defines a set $Q(E)$ of unification equations and a term obtained from t denoted $E(t)$, which are obtained using the algorithm below

> **begin**
> $Q = \emptyset$,
> $s = t$,
> **for** $k = 1$ **to** n **do**
> **begin**
> let E_k be $A\ i = A\ j$,
> **if** $A\ i$ or $A\ j$ is not an address in s
> **then abort,**
> add to Q the equation $s\ A\ i = s\ A\ j$,
> /* this is an equation between *real* terms *not* addresses */
> update s by deleting subterm $s\ A\ j$
> **end,**
> let $E(t)$ be s,
> let $Q(E)$ be Q
> **end**

An E-sequence is *valid* in t if the above algorithm does not abort on input t and E Intuitively, if an E-sequence is not valid it definitely does not describe a standard compaction Even if an E-sequence is valid it does not necessarily describe a standard compaction since the unification equations may not be satisfiable Furthermore, even if a standard compaction is described it does not necessarily end up in a strongly compact term, and hence cannot depict a binding followed by a standard compaction ending up with a standard fact

An E-sequence E is ***satisfiable*** in t if it is valid 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 a ***generic term for t defined by E and ω*** If E defines a generic term for t then this term is a variant of any other generic term defined by E for t

Claim: Let g be a generic term for t defined by an E-sequence E with mgu ω Then, $t\omega =_s g$

Theorem 3: Let I be a standard fact There exists a standard substitution θ such that $t\theta =_s I$ iff there exist a substitution δ and an E-sequence E, inducing a satisfiable $Q(E)$ via mgu ω and a generic $g = E(t)\omega$, such that $t\omega =_s g$, $g\delta = I$ and $\theta = \omega\delta$ is standard

4. The Rewriting Transformation

By considering all possible valid E-sequences on t, the set of all pairs where each pair is of the form (g, ω) of generic terms of t and the mgus generating them, denoted $G(t)$, may be obtained There are better ways for obtaining $G(t)$, but still exponential in the size of t This is not surprising as set matching is NP-hard [KN86] We leave this subject for a subsequent paper

4.1. The first step

We now explain the transformation A rule r of the form

$head \leftarrow t_1, \quad , t_n$ where, w l o.g , t_1 contains *set_of* subterms is transformed into a rule r' of the form

$head \leftarrow funnel\_up\_t_1, t_2, \quad , t_n$

and a set of *permutation rules*

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

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

where $permute\_1\_t_1, \quad , permute\_m\_t_1$ are all the permutations of term t_1 Each such permutation is obtained from t by exchanging positions of arguments of some *set_of* subterms 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 ransforming rule r in P as above For a set of facts S, let S/P be the subset of facts in S whose redicate symbol appears in P Let us refine the ction of satisfaction of a rule body as follows ubstitution θ *satisfies the body* of a rule $\leftarrow t_1, \ldots, t_n$ in a set of facts S, if for $=1, \ldots, n$, there exists $s_i \in S$ such that (i) $\theta = s_i$ if t_i is a funnel_up literal, (ii) $t_i\theta =_i s_i$ if t_i a permute_i literal, (iii) if t is of the form $a=b$ en $a\theta =_{ci} b\theta$, (iv) if t is of the form $a \neq b$ then $\theta \neq_{ci} b\theta$, and otherwise $t_i\theta =_{ci} s_i$

Lemma 4.1: Assume that in the definition of $M(P)$ (1) only standard substitutions are considered, (2) the refined notion of rule body satisfaction is used, and (3) each added fact, which is **not with predicate name prefix** ***funnel_up_***, is standardized Let $\overline{M}(P)$ be the resulting set Then, $\overline{M}(P+funnel)/P =_{ci} M(P)$

4.2. The second step

n the next step of the transformation, each permutation rule $funnel\_up\_t_1 \leftarrow permute\_i\_t_1$ is **deleted** nd replaced with, usually many, *generic rules* btained from $G(permute\_i\_t_1)$ For each pair $g, \omega)$ 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* terms in this example, each ω can be represented by indicating which arguments were identified as equal by ω Once this is done, a standard compaction gets 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 set of new generic rules and no permutation rules

Define $P+generic$ as the resulting program following the transformation Let us further refine the notion of rule body satisfaction by adding "(v) $t_i\theta = s_i$ if t_i is a generic literal," to the definition in the previous section

Lemma 4.2: Suppose that in the definition of $M(P)$ (1) only standard substitutions are considered, (2) the newly refined notion of rule body satisfaction is used, and (3) each added fact, which is **not with predicate name prefix** ***funnel_up_***, is standardized Let $\overline{M}(P)$ be the resulting set Then, $\overline{M}(P+generic)/P =_{ci} M(P)$

4.3. 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 (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))

[]

4.4. 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 Intuitively, a generic rule represents a possible compaction which may be applicable at run-time

(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, see next section

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') One would like to argue, based on Lemma 4 2, that assuming that derived facts, other than those derived for generic rules, are standardized in computing $M(P')$, $M(P')$ may be computed by only considering ordinary matching This argument seems to follow from the fact that once P' is formed, all literals containing ***set_of*** subterms are either funnel_up literals or generic literals

However, there is one delicate point to consider It is still possible that a generic rule will match its generic literal to a standard fact I via δ such that $\theta{=}\omega\delta$ is not standard! In that case we may end up considering non-standard θ's in computing $\overline{M}(P')$ But, if such a θ is used to match ***funnel_up_***t_1 in the body of some r' with ***funnel_up***$t_1\omega\delta$ generated by some generic rule, we still have $t_1\omega =_c$ ***permute_i_***$t_1\omega =_i g$ which implies $t_1\omega =_{ci} g$ which implies $t_1\omega\delta =_{ci} t_1\theta =_{ci} g\,\delta = I$ So, even if such a non-standard θ "satisfies" the body of a rule, the derived ***standard*** ($h\,\theta$) would have been in $M_1(P)$ and so the "extra" facts we generate result in $\overline{M}' = \overline{M}(P) \cup$ extra facts, such that $\overline{M}' =_{ci} M(P)$ Hence, correct query results are obtained by considering the "ordinary" logic programming model for P' with the provision that facts generated by non-generic rules are standardized

5. Optimization

The following techniques apply at step 6 of the rule transformation summary of the previous section We consider original rule r, its modification r', and its funnel-up literal *funnel_up_t* Let GR be the set of MHSB rules generated by the rewriting We shall use m to denote a MHSB rule in GR Let P' be the resulting program.

5.1. Using equalities and inequalities

In some cases it may be determined that certain funnel-up heads in a MHSB rule cannot supply any bindings for which the whole (modified) rule body can succeed in matching all literals, in such cases these heads are disposed of in advance Such cases often involve arithmetic predicates and the predicates $=$ and $\neq$ 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 violating $X \neq john$. 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 certain 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. 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 easily be checked.

5.2. 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 ***unmatchable*** if it cannot match any standard fact I The decision problem as to whether a given term is unmatchable 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 unmatchable and yet not be antiordered, e g , in

$$t = f\,(set\_of\,(1,X), set\_of\,(X,1)),$$

each *set_of* subterm of t by itself can match with a standard fact, yet t cannot We have the following

Observation 5.1: An antiordered term is unmatchable []

So, if a generic literal is antiordered, and hence unmatchable, 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 which determines a total order on terms which when restricted to ground terms reduces to the total order on ground terms defined previously It basically assumes that any order is possible when one of the terms is a variable

```
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 ) Λ i ≤arity (s ) Λ 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
```

$comp = arity(t) \leq arity(s)$,
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. For instance, on *set_of(X, Y, f(Y), f(X))* procedure *precedes* returns **true**. Observe that no matching is possible since, once X an Y are instantiated, we cannot have both X precedes Y and Y precedes X in the total order on terms. However, the above procedure is computationally feasible and detects many cases in which t is antiordered.

5.3. Using Synonyms

Other cases involve optimization techniques 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 we can think about this substitution as 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 predicate (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. Recall that P' is 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 against P (and 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}. Let

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

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

Theorem 4: If the synonym test applied to h_1 and h_2 succeeds, then 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. For example, consider the rule

john_friend(X) ← friends(set_of(X,Y,john)),
X ≠ john, nice(X)

We now examine a MHSB rule, for example rule (4') above. We see that after applying the distinguished substitution $\alpha = \{X / x\}$ to the two heads in rule 4' we obtain $h_{1b} =$ *funnel_up_friends*(*set_of*$(x, x, john)$) and $h_{2b} =$ *funnel_up_friends*(*set_of*$(x, john, john)$). Thus, we get $\theta_1 = \{X/x, Y/x\}$ and $\theta_2 = \{X/x, Y/john\}$. Consequently,

$head_1' = john\_friend(x) =_{ci} john\_friend(x) = head_2'$

and

$body_1' = set\_of(x \neq john, nice(x)) =_{ci} set\_of(x \neq john, nice(x)) = body_2'$. Therefore, *funnel_up_friends*(*set_of*$(X, X, john)$) and *funnel_up_friends*(*set_of*$(X, john, john)$) 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.

6. Multihead Rules

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 data items *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 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 i n 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 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 $\{X_1/Y_1, \quad ,X_n/Y_n\}$ where $X_1, \quad ,X_n$ are distinct variables and $\{X_1, \quad ,X_n\}=\{Y_1, \quad ,Y_n\}$ 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 $B_1, \quad , B_n$, 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* ($B_1, \quad , B_n$)

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

We can extend the definition of $M(P)$ ((respectively, $\overline{M}(P)$) 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* <—*body*, then replacing in P r with $\bar{r}$ result in the same $M(P)$ (respectively, $\overline{M}(P)$ for $\bar{r}'$), where $\bar{r}$ is *head*, *head* θ ← *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 of the previous section 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 idempo tence, 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 model 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'

7. Compiling MHMB rules

In this section we sketch some ideas concerning the compilation of the rewritten program into a target language (e g C, or Prolog) MHSB rules having the same set S of multiheads can be grouped into MHMB rules, where the multi-body part is the collection of distinct bodies and the multihead part is S Thus, a MHMB represents many rules, each formed by a head from the MH part and a body from the MB part This notation presents an opportunity for compiling all these many rules as a single unit The general problem is given a set of bodies, determine an inexpensive sequence of steps to determine all satisfiable bodies and the satisfying substitutions The sequence produced is similar to Prolog-like backtracking 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 Thes e 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!***

8. 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 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 blind alleys explored by the blind search of E-matching In particular we take advantage of the standard representation of facts, the inequality constraints and synonyms

Some of the techniques described, 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 unmatchable, i e cannot match with any standard fact, is still open Additional optimization techniques also seem feasible

Lastly, 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's not very practical For large sets we can resort to using other techniques which rely on set membership tests, this technique is outside the scope of this paper

In many such large sets, many of the *set_of* arguments are variables that appear there 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

Acknowledgment

The authors wish to thank C Beeri for the useful comments received upon reading an early version of this paper They also would like to thank Y Sagiv for his many useful comments

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 8·2, pp 218-246. 1976.

[BNRST87] Beeri, C , S Naqvı, R. Ramakrishnan, O Shmuelı, and S. Tsur *Sets and negatıon ın a Logic Database Language (LDL1)* To appear ın 6th ACM Symposium on Principles of Database Systems, San Diego, CA , 1987

[CM84] Clocksin, W F , and C S Mellısh *Programmıng ın Prolog,* 2n d. Editıon, Sprınger Verlag, 1984.

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

[KN86] Kapur, D., and P. Narendran *NP-Completeness of the Set Unıficatıon and Matchıng Problem* Proc 8th Internatıonal Conference on Automated Deductıon (CADE-8) Oxford, England, July 1986

[KRS84] Korth, H F., M.A Roth, and A Sılberschatz, *A Theory of Non-Fırst-Normal-Form Relatıonal Databases,* 1984.

[KV84] Kuper, G.M , and M Y. Vardı *A new Approach to Database Logıc.* Proc Thırd ACM Symp on Princıples of Database Systems, Waterloo, Canada, 1984

[LC87] Lincoln P , and J Christıan *Adventures ın Assocıative-Commutatıve Unıfication.* MCC Tech. Rept ACA-AI-275-87, Sept 28, 1987

[LLOY84] Lloyd, J W , *Foundatıons of Logıc Programming* Springer Verlag, 1984

[LS76] Lıvesey, M., and A J. Sıekmann, *Unificatıon of A + C -Terms (Bags) and A+C+I-Terms (Sets)* Universıtat Karlsruhe, Fakultat fur Informatık Technıcal Report 5/76, 1976.

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

[RS79] Raulefs, A P , A J. Sıekmann, P Szabo, and E Unverıcht *A Short Survey on the State of the Art in Matchıng and Unification Problems* ACM Sıgsam Bulletın 13,2, pp 14-20, May 1979

[STZ87] Shmuelı, O., Tsur S , and C Zanıolo *Compılatıon of Rules Contaınıng Set Terms ın a Logic Data Language (LDL)* MCC Tech Rept DB-222-87, (revısed)

[STICK81] Stıckel, M E , *A Unıficatıon Algorıthm for Assocıatıve-Commutatıve Functıons* J. of the ACM 28,3, pp 423-434, July 1981

[SZ87] Sacca', D , and C Zanıolo *Implementatıon of Recursıve querıes for a Data Language Based on Pure Horn Clauses* Fourth Internatıonal Conference on Logıc Programmıng, Melbourne, Australıa, 1987

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