Proceedings From the Second International Conference on

EXPERT DATABASE SYSTEMS

LARRY KERSCHBERG, EDITOR

George Mason University

THE BENJAMIN/CUMMINGS PUBLISHING COMPANY, INC.

Redwood City, California • Fort Collins, Colorado
Menlo Park, California • Reading, Massachusetts • New York
Don Mills, Ontario • Wokingham, U.K. • Amsterdam • Bonn
Sydney • Singapore • Tokyo • Madrid • San Juan

Handling Knowledge by its Representative

by

Chiaki Sakama and Hidenori Itoh
Institute for New Generation Computer Technology
Tokyo, Japan

ABSTRACT

This paper presents a method of handling knowledge by its representative under the equivalence relation between literals in a Horn clause program. This method enables us to deal uniformly with knowledge expressed differently and to utilize knowledge dynamically in multiple worlds.

1 Introduction

When it is considered to represent knowledge by Horn logic such as deductive databases, knowledge is represented by a set of Horn clauses with literals. A literal is composed of a list of arguments which denote entities, and a predicate which denotes their relation or property. In such a case, individuals with different names usually denote different entities (*unique name assumption* [Gal 84]), but different predicate names do not always mean different relationships between entities. For example, the literals $parent(john, mary)$, $offspring(mary, john)$ and $child(mary, john)$ denote the same relationship between the entities, *john* and *mary*. This is because the predicate names *child* and *offspring* have an identical relation and *parent* is in a counter relation to them.

To handle such equivalence relations, a simple way is to represent such equivalence relations explicitly by some clauses in a program such as

$parent(X, Y) : -child(Y, X)$,
$child(X, Y) : -offspring(X, Y)$ and
$offspring(X, Y) : -parent(Y, X)$.

However, this way needs at least as many clauses as that of the equivalent literals, and they are somewhat meaningless for deduction. (They will cause an infinite loop in a depth first search such as Prolog since they are mutually recursive.)

[Yoko 86] used such relationships in a logic database by setting some base predicates and defining *higher-order-relations* between those base predicates

and other predicates, then utilizing them in the context of analogical query optimization.

In this paper, such equivalence relations are treated by equivalence classes of literals. That is, for a given set of literals, there are defined some equivalence classes for them and chosen a representative in each class. And then, these representatives are used for the manipulation of Horn clause programs.

Section 2 represents the classification of literals and choice of their representatives, section 3 describes the program transformation using such representatives, and section 4 shows its application to query processing in a knowledge base.

2 Classification of Literals

2.1 Equivalence Relation between Literals

For the sake of brevity, function symbols in terms are omitted in the following discussion, that is, function free literals are assumed. First, the equivalence relation between literals is defined.

Definition 2.1 Suppose a set of literals Λ, the equivalence relations $\varepsilon_1,...,\varepsilon_n$ over Λ are called *worlds*. For $\exists\lambda \in \Lambda$, the set

$$\Phi_\lambda^{\varepsilon_i} = \{\lambda_k \mid \lambda_k \in \Lambda, \lambda_k \varepsilon_i \lambda\}$$

is called an *equivalence class* of *representative* λ in ε_i.

In the world ε_i, Λ is partitioned into disjoint subsets of equivalence classes; such partitioning is called *classification* of Λ in ε_i. □

Especially, when

$$\forall\lambda_k, \lambda_l \in \Lambda, \quad if \quad \lambda_k \varepsilon_i \lambda_l \quad then \quad \lambda_k \varepsilon_j \lambda_l$$

holds, ε_i is called *stronger* than ε_j (or ε_j is *weaker* than ε_i).

Example 2.1 When a set of literals

$$\Lambda = \{parent(X,Y), child(X,Y), child\_in\_law(X,Y)\}$$

is given, Λ is classified into the equivalence classes in the world, say, *blood* :

$$\Phi_{parent}^{blood} = \{parent(X,Y), child(Y,X)\}$$
$$\Phi_{child\_in\_law}^{blood} = \{child\_in\_law(X,Y)\}$$

(Note that some appropriate renaming of variables are done in the equivalence class, and the arguments are missed to denote the representative if there is no ambiguity.)

While, in the world, say, *family*, Λ is classified as follows:

$$\Phi_{parent}^{family} = \{parent(X,Y), child(Y,X), child\_in\_law(Y,X)\}$$

That is, the world *blood* is stronger than *family*. □

The equivalence relations are known to make a complete lattice under such partial ordering over relations.

Suppose an equivalence class of literals which include some variables in a world. In this case, if there exist some instances of those literals, then they also make another equivalence class or a subset of some equivalence class in the world. For example, by applying a substitution $\theta = \{john/X, mary/Y\}$ to the above class Φ^{blood}_{parent}, an instance

$$\Phi^{blood}_{parent}\theta = \{parent(john, mary), child(mary, john)\}$$

also make an equivalence class with the representative, *parent*.

In this case, the representative of the instance equivalence class agree with that of the original equivalence class. However, it is not such straightforward in general to choose a representative in an equivalence class so that its instance class also has its instance representative.

Next, a criterion to choose such a representative from an equivalence class is discussed.

2.2 Choice of Representative

In the following of this paper, when it is discussed in a single world, the denotation of the world is often omitted, that is, Φ^{ϵ}_{ϕ} are simply denoted as Φ_{ϕ}. And Φ denotes the equivalence class whose representative is not still chosen.

Suppose a set of equivalence classes $\mathbf{E}$, then the representatives of each class are chosen according to the following criterion.

1. If an equivalence class $\Phi \in \mathbf{E}$ has no unifiable elememt with any other classes $\Psi \in \mathbf{E}$ then choose one element $\phi \in \Phi$ as a representative of Φ.
2. Otherwise, choose one element $\phi \in \Phi$ which is unifiable with elements of other classes.

According to this criterion, a representative of an equivalence class is to be chosen so that its instance class also has the instance representative.

Example 2.2 Suppose the equivalence classes:

$$\Phi = \{gt(X, Y), lt(Y, X)\}$$
$$\Psi = \{gt(X, 0), lt(0, X), positive(X)\}$$

Here, $gt(X, Y)$, $lt(Y, X)$ means that X is greater than Y, Y is lower than X, and $positive(X)$ means that X is a positive number.

Now consider the substitutions $\theta_1 = \{1/X, 0/Y\}$ for Φ and $\theta_2 = \{1/X\}$ for Ψ. Then the instances of Φ and Ψ with these substitutions are as follows.

$$\Phi\theta_1 = \{gt(1, 0), lt(0, 1)\}$$
$$\Psi\theta_2 = \{gt(1, 0), lt(0, 1), positive(1)\}$$

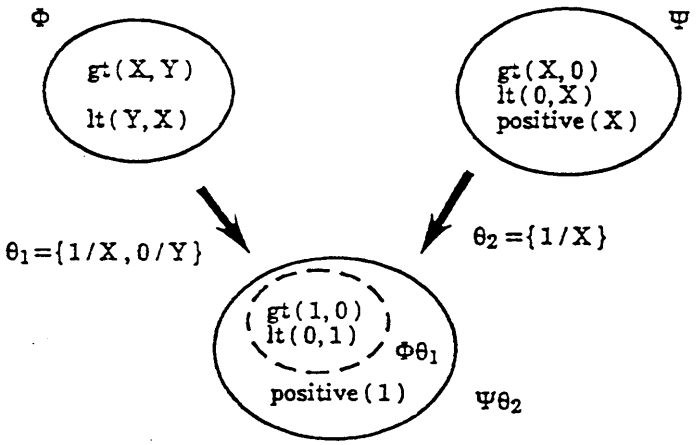

Figure 1. Example 2.2

In this case, $\Phi\theta_1$ is included in $\Psi\theta_2$ which is an equivalence class for these instances.

To choose a representative of $\Psi\theta_2$ uniquely, according to the criterion, the representatives of Φ and Ψ must be either $\phi = gt(X,Y)$ and $\psi = gt(X,0)$, or $\phi = lt(Y,X)$ and $\psi = lt(0,X)$, but $positive(X)$ cannot be chosen as the representative of Ψ. Then the representative of $\Psi\theta_2$ becomes $gt(1,0)$ or $lt(0,1)$ (Figure 1). □

Although this method requires to check the unifiability of each element among the equivalence classes, the choice of representatives would be done once for all in each class, and re-examination is needed only when there happens some updation to the classes.

However, there is a case when a representative of an equivalence class cannot be chosen uniquely by that criterion.

Example 2.3 Suppose the equivalence classes:

$\Phi = \{add(X,Y,Z), minus(Z,X,Y)\}$
$\Psi = \{multiply(X,Y,Z), divide(Z,X,Y)\}$
$\Omega = \{add(X,X,Y), multiply(X,2,Y)\}$

Here, $add(X,Y,Z)$, $minus(X,Y,Z)$, $multiply(X,Y,Z)$ and $divide(X,Y,Z)$ means that $X+Y=Z$, $X-Y=Z$, $X*Y=Z$ and $X/Y=Z$, respectively.

In this case, as the representative of Ω, $add(X,X,Y)$ can be chosen as a unifiable element with $add(X,Y,Z)$ in Φ, while $multiply(X,2,Y)$ can also be chosen as a unifiable element with $multiply(X,Y,Z)$ in Ψ. □

This happened because there is no element in Ω which is unifiable with both of the elements in Φ and Ψ, so the representative of Ω cannot be chosen uniquely by the criterion.

There may be considered some method to treat such a case, but in the

following discussion it is assumed that when the representatives of some equivalence classes cannot be chosen uniquely by the criterion, we define no equivalence relations between those literals. That is, each of those literals are considered representatives for themselves.

In the above example, the equivalence classes and representatives are:

$\Phi_{add} = \{add(X,Y,Z)\}$
$\Phi_{minus} = \{minus(X,Y,Z)\}$
$\Phi_{multiply} = \{multiply(X,Y,Z)\}$
$\Phi_{divide} = \{divide(X,Y,Z)\}$.

Thus they are treated independently as usual.

3 Program Transformation using Representatives

3.1 Representative Program

Suppose there is given a function free Horn clause program and some equivalence classes and their representatives for the literals appearing in the program. Then, replacing all of those literals in the program by their representatives, the original program is transformed into a program which only includes the representative for those literals. Such a transformation is called a *representative transformation* and the transformed program is called a *representative program.*

Example 3.1 Suppose the following program S.

$S = \{\ ancestor(X,Y) : -parent(X,Y).$
$\quad ancestor(X,Y) : -parent(X,Z), ancestor(Z,Y).$
$\quad parent(john, mary).$
$\quad child(lisa, mary).\ \}$

When equivalence classes $\Phi_{ancestor}$ and Φ_{parent} are defined as:

$\Phi_{ancestor} = \{ancestor(X,Y)\}$
$\Phi_{parent} = \{parent(X,Y), child(Y,X)\}$

then S is transformed into the following representative program S':

$S' = \{\ ancestor(X,Y) : -parent(X,Y).$
$\quad ancestor(X,Y) : -parent(X,Z), ancestor(Z,Y).$
$\quad parent(john, mary).$
$\quad parent(mary, lisa).\ \}$ □

In the above example, $ancestor(john, lisa)$ is deduced from S' but not from S. This means that by transforming the original program into the representative program, the result of the deduction does not depend on the predicate names but on their meanings in the original program.

Example 3.2 Suppose the following programs S_1 and S_2.

$S_1 = \{\ ancestor(X,Y) : -parent(X,Y).$
$ancestor(X,Y) : -parent(X,Z), ancestor(Z,Y).$
$parent(X,Y) : -child(Y,X).$
$child(mary, john).$
$child(lisa, mary).\ \}$

$S_2 = \{\ descendant(X,Y) : -offspring(X,Y).$
$descendant(X,Y) : -offspring(X,Z), descendant(Z,Y).$
$offspring(jack, lisa).$
$offspring(lisa, mary).\ \}$

When equivalence classes $\Phi_{ancestor}$ and Φ_{parent} are defined as follows:

$\Phi_{ancestor} = \{ancestor(X,Y), descendant(Y,X)\}$
$\Phi_{parent} = \{parent(X,Y), child(Y,X), offspring(Y,X)\}$

S_1 and S_2 are transformed into the following representative programs S_1' and S_2', respectively.

$S_1' = \{\ ancestor(X,Y) : -parent(X,Y).$
$ancestor(X,Y) : -parent(X,Z), ancestor(Z,Y).$
$parent(X,Y) : -parent(X,Y).$
$parent(john, mary).$
$parent(mary, lisa).\ \}$

$S_2' = \{\ ancestor(Y,X) : -parent(Y,X).$
$ancestor(Y,X) : -parent(Z,X), ancestor(Y,Z).$
$parent(lisa, jack).$
$parent(mary, lisa).\ \}$

In this case, $ancestor(john, jack)$ can be deduced from the union of the programs $S_1' \cup S_2'$, but not from $S_1 \cup S_2$. (Note that such a transformed program has some redundant clauses. This problem is discussed later.) □

This example shows that different programs which are written independently can be treated uniformly by transforming them into representative programs. In other words, the representatives are considered as a global expression for the local programs, and through such a transformation one local program can utilize the knowledge represented in another local programs. (Here, the same literals are assumed to have the same meaning between different programs.)

3.2 Program Transformation in Multiple Worlds

In this subsection, the representative transformation of program is considered in multiple worlds.

Example 3.3 Suppose the following program S.

$$S = \{\ stranger(X) : -foreigner(X).$$
$$american(reagan).$$
$$japanese(nakasone).\ \}$$

Assume the following equivalence classes in a world, *japan* :

$$\Phi^{japan}_{stranger} = \{stranger(X)\}$$
$$\Phi^{japan}_{foreigner} = \{foreigner(X), american(X)\}$$
$$\Phi^{japan}_{japanese} = \{japanese(X)\}$$

In this case, S is transformed into S^{japan} such as:

$$S^{japan} = \{\ stranger(X) : -foreigner(X).$$
$$foreigner(reagan).$$
$$japanese(nakasone).\ \}$$

and $stranger(reagan)$ is deduced from S^{japan}.

While consider another equivalence classes in a world, *usa* :

$$\Phi^{usa}_{stranger} = \{stranger(X)\}$$
$$\Phi^{usa}_{american} = \{american(X)\}$$
$$\Phi^{usa}_{foreigner} = \{foreigner(X), japanese(X)\}$$

In this case, S is transformed into S^{usa} such as:

$$S^{usa} = \{\ stranger(X) : -foreigner(X).$$
$$american(reagan).$$
$$foreigner(nakasone).\ \}$$

and $stranger(nakasone)$ is deduced from S^{usa}. □

Example 3.4 Suppose the following program S.

$$S = \{\ path(X,Y) : -edge(X,Y).$$
$$path(X,Y) : -path(X,Z), path(Z,Y).$$
$$edge_{s1}(a,b).$$
$$edge_{s1}(b,c).$$
$$edge_{s2}(b,d).\ \}$$

This program states that there is a path if there are some edges between the nodes, and $edge_{si}(X,Y)$ means that there is an edge between X and Y in a state, s_i.

First, to find paths in the state s_1, assume such equivalence classes:

$$\Phi^{s1}_{path} = \{path(X,Y)\}$$
$$\Phi^{s1}_{edge} = \{edge(X,Y), edge_{s1}(X,Y)\}$$
$$\Phi^{s1}_{edge_{s2}} = \{edge_{s2}(X,Y)\}$$

In this case, S is transformed into S^{s1}, such as:

$$
\begin{aligned}
S^{s1} = \{\ & path(X,Y) : -edge(X,Y). \\
& path(X,Y) : -path(X,Z), path(Z,Y). \\
& edge(a,b). \\
& edge(b,c). \\
& edge_{s2}(b,d).\ \}
\end{aligned}
$$

then $path(a,b), path(b,c)$ and $path(a,c)$ are deduced from S^{s1}.

Next, suppose to find paths in the state $s_1 + s_2$, and assume the weaker equivalence classes as follows.

$$
\begin{aligned}
\Phi_{path}^{s1+s2} &= \{path(X,Y)\} \\
\Phi_{edge}^{s1+s2} &= \{edge(X,Y), edge_{s1}(X,Y), edge_{s2}(X,Y)\}
\end{aligned}
$$

In this case, S or S^{s1} is transformed into S^{s1+s2} such as:

$$
\begin{aligned}
S^{s1+s2} = \{\ & path(X,Y) : -edge(X,Y). \\
& path(X,Y) : -path(X,Z), path(Z,Y). \\
& edge(a,b). \\
& edge(b,c). \\
& edge(b,d).\ \}
\end{aligned}
$$

then $path(b,d)$ and $path(a,d)$ are deduced from S^{s1+s2} in addition to the results of S^{s1}. □

As shown in the above examples, the same program can be interpreted in different way, by changing the equivalence relations in a program according to the world where the program is interpreted.

The multiple worlds (or possible worlds) mechanism in knowledge representation is also discussed in [Moo 77] and [Naka 84]. In these language, the worlds are expressed in the axioms and provides the ability of conditional reasoning.

Here, different equivalence relations are considered multiple worlds and they are used for the program generation in each world. (Note that when more than one program is processed, such as in *Example 3.2*, they must be transformed into the representative programs in the same world.)

3.3 Semantics of Representative Program

This subsection considers the declarative meaning of a representative program. Suppose a Horn clause program S, a representative transformation of literals in a world T, and the transformed representative program $T(S)$. Then the following proposition holds.

Proposition 3.1 If a proposition p is deducible from S then $T(p)$ is deducible from $T(S)$.

Proof: The proof is by induction on the length of the deduction steps.

First, suppose that p is deducible from S in 0-step. In this case, $p \in S$ then $T(p) \in T(S)$, hence $T(p)$ is deducible from $T(S)$ in 0-step.

Next, assume that if p is deducible from S in n-step then $T(p)$ is also deducible from $T(S)$ in n-step. Now suppose p is deducible from S in $n+1$-step as follows:

$$\frac{\Gamma_1 \vee \Delta_1 \quad \Gamma_2 \vee \neg\Delta_2}{\Gamma_1\theta \vee \Gamma_2\theta}$$

Here, $\Gamma_1\theta \vee \Gamma_2\theta = p$, Γ_1, Γ_2 denotes a disjunction of literals, Δ_1, Δ_2 denotes a literal, and θ is an *mgu* of Δ_1 and Δ_2 where $\Delta_1\theta = \Delta_2\theta$.

From the assumption, there is a deduction from $T(S)$ in n-step, as follows.

$$\frac{\vdots}{T(\Gamma_1 \vee \Delta_1)} \quad \frac{\vdots}{T(\Gamma_2 \vee \neg\Delta_2)}$$

Now, $T(\Gamma_1 \vee \Delta_1) = T(\Gamma_1) \vee T(\Delta_1)$ and $T(\Gamma_2 \vee \neg\Delta_2) = T(\Gamma_2) \vee \neg T(\Delta_2)$ holds, and $T(\Delta_1)\theta = T(\Delta_2)\theta$ is yielded by $\Delta_1\theta = \Delta_2\theta$. Then the following deduction can be made at $n+1$-step.

$$\frac{T(\Gamma_1) \vee T(\Delta_1) \quad T(\Gamma_2) \vee \neg T(\Delta_2)}{T(\Gamma_1)\theta \vee T(\Gamma_2)\theta}$$

Since $T(\Gamma_1)\theta \vee T(\Gamma_2)\theta = T(\Gamma_1\theta \vee \Gamma_2\theta) = T(p)$ holds, $T(p)$ is deduced from $T(S)$ in $n+1$-step. □

Note that the reverse of the above theorem does not hold, because $\Delta_1\theta = \Delta_2\theta$ is not yielded by $T(\Delta_1)\theta = T(\Delta_2)\theta$ in general. (For example, consider the case where $\Delta_1 = parent(X, Y)$, $\Delta_2 = child(Y, X)$ and $T(\Delta_1) = T(\Delta_2) = parent(X, Y)$.)

Suppose the least Herbrand model of S, $M(S)$, and that of $T(S)$, $M(T(S))$. *Theorem 3.1* denotes the following relationship.

$$T(M(S)) \subseteq M(T(S)) \tag{1}$$

Now suppose that T^{-1} is mapping from a representative to all of the elements of its equivalence class, then from (1), the following relation holds.

$$M(S) \subseteq T^{-1}(M(T(S))) \tag{2}$$

(2) denotes the model, which is obtained by applying reverse representative transformation to the least model of a representative program, includes the least model of the original program. That is, such a transformation may generate some consequences which are not deduced from the original program but are deducible after the transformation, as well as it preserves the consequences of the original program.

3.4 Program Optimization

Generally, when a original program includes some literals which are in equivalence relationships, the transformed representative program tends to have some redundancy. Recall in *Example 3.2*, the transformed representative program, $S_1' \cup S_2'$, included four redundant clauses:

$$parent(X,Y) : -parent(X,Y).$$
$$parent(mary, lisa).$$
$$ancestor(Y,X) : -parent(Y,X).$$
$$ancestor(Y,X) : -parent(Z,X), ancestor(Y,Z).$$

Redundant clauses in a representative program are those which are expressed differently in the original program, but become redundant after transformation.

To optimize the representative program, it is desirable to remove redundant clauses from the program. However, the optimization of logic programs is not straightforward, because the equivalence of logic programs is known to be undecidable in general, except for assuming several conditions [Mah 86], [Nau 86], and [Sag 87]. Here, the *deletion strategy* [Cha 73] is assumed for the program optimization for its simplicity. The deletion strategy is as follows:

1. If a clause C includes the same literal both in the head and the body of C, then C is a *tautology* and is deleted.
2. If a clause C *subsumes* another clause C', (i.e. for some substitution σ, $C\sigma \subseteq C'$) then C' is deleted.

Applying this strategy to the above example, the clause,

$$parent(X,Y) : -parent(X,Y).$$

is a tautology, hence it is deleted.

While the two clauses,

$$ancestor(Y,X) : -parent(Y,X).$$
$$parent(mary, lisa).$$

are subsumed by the clauses,

$$ancestor(X,Y) : -parent(X,Y).$$
$$parent(mary, lisa).$$

respectively, hence they are deleted, too.

However, the equivalence of two clauses,

$$ancestor(X,Y) : -parent(X,Z), ancestor(Z,Y).$$
$$ancestor(Y,X) : -parent(Z,X), ancestor(Y,Z).$$

will not come out of the strategy and remain in the transformed program.

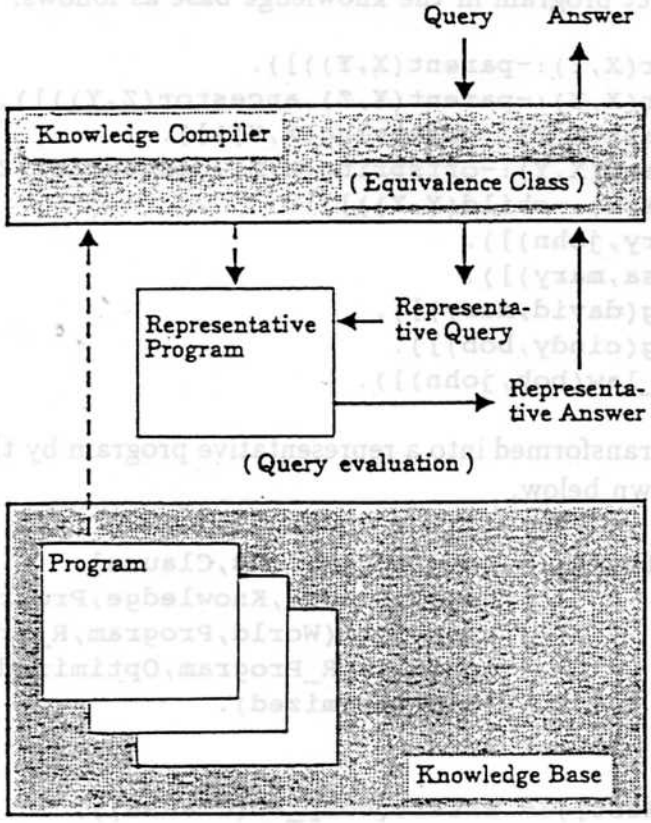

Figure 2. Query processing in knowledge base

(Such an equivalence of clauses will be shown by using induction, but it beyonds the study in this paper.)

4 Query Processing in Representative Program

This section presents the application of the representative transformation to the query processing in a knowledge base. A query which is composed of representative literals is called a *representative query* and its answer tuples are called *representative answer*.

Figure 2 shows query processing in a knowledge base. There are stored Horn clause programs in the knowledge base, and the equivalence relations are defined in the *knowledge compiler*.

Firstly, the knowledge compiler transforms the object program into a representative program using the equivalence relations and optimizes the representative program using the deletion strategy.

Secondly, a query for the program is also transformed into a representative query by the compiler and evaluated in the representative program.

Thirdly, the representative answer, which is the result of the evaluation, are transformed into the answer tuples for the given query by the compiler again, and output. This process is shown by an example.

Suppose the object program in the knowledge base as follows.

```
kb([(ancestor(X,Y):-parent(X,Y))]).
kb([(ancestor(X,Y):-parent(X,Z),ancestor(Z,Y))]).
kb([(descendant(X,Y):-offspring(X,Y))]).
kb([(descendant(X,Y):-offspring(X,Z),descendant(Z,Y))]).
kb([(parent(X,Y):-child(Y,X))]).
kb([child(mary,john)]).
kb([child(lisa,mary)]).
kb([offspring(david,mary)]).
kb([offspring(cindy,bob)]).
kb([child_in_law(bob,john)]).
```

This program is transformed into a representative program by the knowledge compiler as shown below,

```
k_compile(World,KB):- Knowledge=..[KB,Clause],
                      bagof(Clause,Knowledge,Program),
                      transform(World,Program,R_Program),
                      optimize(R_Program,Optimized),
                      load(Optimized).

load([]).
load([Clause|Rest]):- assert(comp_kb(Clause)),
                      load(Rest).

transform(_,[],[]).
transform(World,[[Clause]|S],[[R_Clause]|RS]):-
                      rewrite(World,Clause,R_Clause),
                      transform(World,S,RS).

rewrite(World,(P:-Q),(RP:-RQ)):- equivalent(World,P,RP),
                                 rewrite(World,Q,RQ).
rewrite(World,(P,Q), (RP,RQ) ):- equivalent(World,P,RP),
                                 rewrite(World,Q,RQ).
rewrite(World,P,      RP     ):- equivalent(World,P,RP).
```

where *k_compile*(*World*, *KB*) means to compile the knowledge, *KB* in the *World*. That is, after collecting clauses in the program, *transform*(*World*, *Program*, *R_Program*) transforms them into a representative program, and then *optimize*(*R_Program*, *Optimized*) optimizes the transformed program using the deletion strategy. (Since the process of *optimize* is a little long, details are omitted here.)

Here, the equivalence relation of literals and their representatives are defined as:

```
equivalent(blood,child(X,Y),parent(Y,X)).
equivalent(blood,offspring(X,Y),parent(Y,X)).
equivalent(blood,descendant(X,Y),ancestor(Y,X)).
```

```
equivalent(family,child(X,Y),parent(Y,X)).
equivalent(family,offspring(X,Y),parent(Y,X)).
equivalent(family,child_in_law(X,Y),parent(Y,X)).
equivalent(family,descendant(X,Y),ancestor(Y,X)).

equivalent(World,Undefined,Undefined).
```

where *equivalent*(*World*, *Literal*, *Representative*) defines the equivalence relation between the *Literal* and its *Representative* in the *World*. Further, *equivalent*(*World*, *Undefined*, *Undefined*) means that literals which have no definition of equivalence relation are evaluated without transformation. This means that the equivalence relationship of literals which appear in the program need not necessarily to be defined.

Then, by the compilation in a world, *blood*:

```
?- k_compile(blood,kb).
```

the program is transformed into the following representative program.

```
comp_kb([(ancestor(A,B):-parent(A,B))]).
comp_kb([(ancestor(A,B):-parent(A,C),ancestor(C,B))]).
comp_kb([(ancestor(A,B):-parent(C,B),ancestor(A,C))]).
comp_kb([parent(john,mary)]).
comp_kb([parent(mary,lisa)]).
comp_kb([parent(mary,david)]).
comp_kb([parent(bob,cindy)]).
comp_kb([child_in_law(bob,john)]).
```

While, in a world, *family*:

```
?- k_compile(family,kb).
```

the transformed representative program is as follows:

```
comp_kb([(ancestor(A,B):-parent(A,B))]).
comp_kb([(ancestor(A,B):-parent(A,C),ancestor(C,B))]).
comp_kb([(ancestor(A,B):-parent(C,B),ancestor(A,C))]).
comp_kb([parent(john,mary)]).
comp_kb([parent(mary,lisa)]).
comp_kb([parent(mary,david)]).
comp_kb([parent(bob,cindy)]).
comp_kb([parent(john,bob)]).
```

As shown above, two different programs are generated here in each world. Next, the process for query evaluation is as follows:

```
evaluate(World,Query):- rewrite(World,Query,RepQ),
                        demo(RepQ),write(Query),nl,fail.
evaluate(_,     _     ).

demo((P,Q)):- !,demo(P),demo(Q).
demo( P   ):- comp_kb([P]).
demo( P   ):- comp_kb([(P:-Q)]),demo(Q).
```

where *evaluate*(*World*, *Query*) means evaluation of the *Query* in the *World*. That is, a query is transformed into a representative query in a world, evaluated in the representative program, and output as the answer tuples for the given query. Here, *demo*(*Query*) is a meta interpreter for the *Query*.

Now suppose a query for the compiled knowledge in the world, *blood*:

```
?- evaluate(blood,descendant(X,Y)).
```

the result of the evaluation is as follows.

```
 descendant(mary,john)
 descendant(lisa,mary)
 descendant(david,mary)
 descendant(cindy,bob)
 descendant(lisa,john)
 descendant(david,john)
*descendant(lisa,john)
*descendant(david,john)
```

While suppose a query for the compiled knowledge in the world, *family*:

```
?- evaluate(family,descendant(X,Y)).
```

the result of the evaluation is as follows.

```
 descendant(mary,john)
 descendant(lisa,mary)
 descendant(david,mary)
 descendant(cindy,bob)
 descendant(bob,john)
 descendant(lisa,john)
 descendant(david,john)
 descendant(cindy,john)
*descendant(lisa,john)
*descendant(david,john)
*descendant(cindy,john)
```

As a result, different answer tuples are obtained in each world. Note that some redundant tuples (marked with $*$) are deduced in the above because they are beyond the deletion strategy.

In this way, when an equivalence relation and its representative for a query and a program are given, a query can be evaluated by transforming it into a representative query. As stated in the previous section, some answer tuples for the query can be obtained only through the representative transformation of the query and the program.

5 Concluding Remarks

This paper presented a method of handling knowledge by its representative in a knowledge base represented by Horn logic. It enables us to handle equivalent knowledge uniformly, and then some local programs can communicate their knowledge with each other using the common representatives in a world. Further, a program is transformed into some local programs by changing equivalence relations in multiple worlds. When it is applied to query processing in a knowledge base, a knowledge compiler plays a part in the transformation and realizes a flexible question answering system.

In this paper, a function free Horn program is assumed as is usual in the deductive database, but to extend it to non-Horn program such as stratified database, further discussion is needed.

Acknowledgments

We would like to thank Kazumasa Yokota for suggesting several improvements of the idea, and are grateful to Mikio Yoshida(IBM), Ko Sakai, Yuji Matsumoto and other colleagues at ICOT for their helpful comments on an earlier version of this paper.

References

[Cha 73] Chang,C.L. and Lee,C.T.L.: "Symbolic Logic and Mechanical Theorem Proving", *Academic Press*, 1973.

[Gal 84] Gallaire,H., Minker,J. and Nicolas,J.M.: "Logic and Databases: A Deductive Approach", *ACM Computing Surveys*, Vol.16, No.2, pp.153-185, 1984.

[Mah 86] Maher,M.J.: "Equivalences of Logic Programs", *Proc. of 3rd ICLP*, pp.410-424, 1986.

[Moo 77] Moore,R.C.: "Reasoning About Knowledge and Action", *Proc. of 5th IJCAI*, pp.223-227, 1977.

[Naka 84] Nakashima,H.: "Knowledge Representation in Prolog/KR", *Proc. of SLP'84*, pp.126-130, 1984.

[Nau 86] Naughton,J.F.: "Redundancy in Function-Free Recursive Rules", *Proc. of SLP'86*, pp.236-245, 1986.

[Sag 87] Sagiv,Y.: "Optimizing Datalog Programs", *Proc. of 6th PODS*, pp.349-362, 1987.

[Yoko 86] Yokomori,T.: "On Analogical Query Processing in Logic Database", *Proc. of 12th VLDB*, pp.376-383, 1986.