Amalgamating Knowledge Bases, II: Algorithms, Data Structures, and Query Processing*

Sibel Adalı and **V.S. Subrahmanian**
Department of Computer Science
Institute for Advanced Computer Studies &
Institute for Systems Research
University of Maryland
College Park, Maryland 20742.
{sibel, vs}@cs.umd.edu

Abstract

Integrating knowledge from multiple sources is an important aspect of automated reasoning systems. In the first part of this series of papers, we presented a uniform declarative framework, based on *annotated logics*, for amalgamating multiple knowledge bases when these knowledge bases (possibly) contain inconsistencies, uncertainties, and non-monotonic modes of negation. We showed that annotated logics may be used, with some modifications, to *mediate* between different knowledge bases. The multiple knowledge bases are amalgamated by embedding the individual knowledge bases into a lattice. In this paper, we briefly describe an SLD-resolution based proof procedure that is sound and complete w.r.t. our declarative semantics. We will then develop an `OLDT`-resolution based query processing procedure, `MULTI_OLDT`, that satisfies two important properties: (1) *efficient reuse* of previous computations is achieved by maintaining a table – we describe the structure of this table and show that table operations can be efficiently executed, and (2) *approximate, interruptable query answering* is achieved, i.e. it is possible to obtain an "intermediate, approximate" answer from the `QPP` by interrupting it at any point in time during its execution. The design of the `MULTI_OLDT` procedure will include: (1) development of data structures for tabling (substitution, truth value) pairs, and (2) the development of algorithms to incrementally and efficiently update the table.

1 Introduction

Complex reasoning tasks in the real world utilize information from a multiplicity of sources. These sources may represent data and/or knowledge about different aspects of a problem in a number of ways. Wiederhold and his colleagues [37, 38] have proposed the concept of a *mediator* – a device that will express how such an integration is to be achieved.

This is the second in a series of papers developing the theory and practice of federated databases. In Part I of this series of papers, we developed a language for expressing mediators, and reasoning with them. In particular, we showed that an extension of the "generalized annotated program" (`GAP`)

*This work was supported by the Army Research Office under Grant Nr. DAAL-03-92-G-0225 and by the Air Force Office of Scientific Research under Grant Nr. F49620-93-1-0065, and by ARPA Order Nr. A716 administered by Rome Labs under contract F30602-93-C-0241. NOTE TO REFEREES: Appendix A contains material that can be removed from the paper after acceptance.

paradigm of Kifer and Subrahmanian [20] may be used to express mediators. We defined the concept of the "amalgam" of "local" databases $DB_1, \ldots, DB_n$ with a mediator or supervisory database, S, and proved a number of results linking the semantics of the local databases with the semantics of the amalgam.

The primary aim of this paper is the *development of query processing procedures* (QPPs, for short) that possess various desirable properties. We will first develop a resolution-based QPP and show it to be sound and complete. However, it is well known that resolution proof procedures are notoriously inefficient, often solving previously solved goals over and over again. OLDT-resolution, due to Tamaki and Sato [34] is a technique which caches previously derived solutions in a table. The theory and implementation of OLDT has been studied extensively by several researchers including Seki [28, 27] and Warren and his colleagues [9, 10]. Furthermore, it is known that OLDT and magic set computations [5, 6, 26] are essentially equivalent, though they differ in many (relatively minor) details. We will use the OLDT technique as our starting point, and extend it as follows:
(1) *Multiple Databases:* As different databases may provide different answers to the same query, OLDT-resolution needs to be modified to handle a multiplicity of (possibly mutually incompatible) answers to the same query.
(2) *Uncertainty and Time:* Previous formulations of OLDT-resolution did not handle time and uncertainty. We will show how temporal and uncertain answers can be smoothly incorporated into the OLDT paradigm.
(3) *Approximate, Interruptable Query Answering:* In some situations, the user may wish to interrupt the execution of the query processing procedure and ask for a "tentative answer." This kind of feature becomes doubly important when databases contain uncertain and temporal information. When processing a query Q such as "Is the object O at location L an enemy aircraft ?," it is desirable that uncertainty estimates of the truth of this query be revised upwards in a monotonic fashion as the QPP spends more and more time performing inferences. Thus if the user interrupts the QPP's execution at time t and asks "What can you tell me about query Q ?," the KB should be able to respond with an answer of the form: "I'm not done yet, but at this point I can tell you that Q is true with certainty 87% or more."
(4) *Table Management:* Relatively little work has been done on the development of data structures for managing OLDT-tables (cf. Warren [9, 10]). When a single database with neither uncertainty nor time is considered, the structure of the OLDT-table can be relatively simple. However, when multiple database operations, uncertainty estimates (that are constantly being revised), and temporal reasoning are being performed simultaneously, the management of the OLDT-table becomes a significant issue. We will develop data structures and algorithms to efficiently manage the OLDT-table.

Our query processing procedure, called MULTI_OLDT, incorporates all the above features and is described in detail in this paper. In particular, we prove that MULTI_OLDT is a sound and complete query processing procedure. Restricted termination results are also established.

The paper is organized as follows; in Section 3, we provide two examples motivating our work. These examples will be used throughout the paper to illustrate various definitions, data structures, and algorithms. Section 4 contains a brief description of a resolution-style proof procedure including soundness and completeness results. The MULTI_OLDT procedure is described in detail in Section 5 – in particular, this section contains details on the organization of the OLDT-table. We compare our results with relevant work by other researchers in Section 6.

2 Preliminaries

In this section, we give a quick overview of GAPs and the amalgamation theory developed in the first of this series of papers [32].

2.1 Overview of GAPs

The GAP framework proposed in [20] assumes that we have a set $\mathcal{T}$ of truth values that forms a complete lattice under an ordering $\preceq$. For instance, $(\mathcal{T}, \preceq)$ may be any one of the following:

(1) *Fuzzy Values:* We can take $\mathcal{T} = [0,1]$ – the set of rea l numbers between 0 and 1 (inclusive) and $\preceq$ to be the usual $\leq$ ordering on reals.
(2) *Time Points:* We can take $\mathcal{T}$ to be the set TIME $= 2^{\mathbf{R}^+}$ where $\mathbf{R}^+$ is the set of non-negative real numbers , $2^{\mathbf{R}^+}$ is the power-set of the reals, and $\preceq$ is the inclusion ordering.
(3) *Fuzzy Values + Time:* We could take $\mathcal{T} = [0,1] \times$TIME and take $\preceq$ to be the ordering: $[u_1, T_1] \preceq [u_2, T_2]$ iff $u_1 \leq u_2$ and $T_1 \subseteq T_2$. Here u_1, u_2 are real numbers in the $[0,1]$ interval and T_1, T_2 are sets of real numbers.
(4) *Four-Valued Logic:* Four valued logic [7, 19] uses the truth values FOUR $= \{\bot, \mathbf{t}, \mathbf{f}, \top\}$ ordered as follows: $\bot \preceq x$ and $x \preceq \top$ for all $x \in$ FOUR. In particular, $\mathbf{t}$ and $\mathbf{f}$ are not comparable relative to this ordering. [7] and [19] show how this FOUR-valued logic may be used to reason about databases containing inconsistencies.

This is only a small sample of what $\mathcal{T}$ could be. Using the elements of $\mathcal{T}$, as well as variables ranging over $\mathcal{T}$ (called annotation variables), and preinterpreted functions of arity $n \geq 1$ on $\mathcal{T}$ (called annotation functions), it is possible to recursively define an *annotation term* as follows: (1) any member of $\mathcal{T}$ is an annotation term, (2) any annotation variable is an annotation term, and (3) if f is an n-ary annotation function and $t_1, \ldots, t_n$ are annotation terms, then $f(t_1, \ldots, t_n)$ is an annotation term. For instance, if $\mathcal{T} = [0,1]$, and $+, *$ are preinterpreted annotation functions defined in the usual ways, and V is an annotation variable, then $(V+1) * 0.5$ is an annotation term. Strictly speaking, we should write this in prefix notation as: $*(+(V,1), 0.5)$, but we will often abuse notation when the meaning is clear from context.

If A is an atom (in the usual sense of logic), and μ is an annotation, then $A : \mu$ is an annotated atom. For example, when considering $\mathcal{T} = [0,1]$, the atom $broken(c_1) : 0.75$ may be used to say: "there is at least a 75% degree of certainty that component c_1 is broken." If $\mathcal{T} = [0,1] \times$ TIME, then annotations are pairs, and an annotated atom like $at\_robot(3,5) : [0.4, \{1,2,3\}]$ says that at each of the time points $1, 2, 3$, there is at least a 40% certainty that the robot is at xy-coordinates $(3,5)$.

An annotated clause is a statement of the form:

$$A_0 : \mu_0 \quad \leftarrow \quad A_1 : \mu_1 \,\&\, \ldots \,\&\, A_n : \mu_n$$

where: (1) each $A_i : \mu_i$, $0 \leq i \leq n$ is an annotated atom, and (2) for all $1 \leq j \leq n$, μ_j is either a member of $\mathcal{T}$ or is an annotation variable, i.e. μ_j contains no annotation functions. In other words, annotation functions can occur in the heads of clauses, but not in the clause bodies.

Kifer and Subrahmanian developed a formal model theory, proof theory, and fixpoint theory for GAPs that accurately captures the above-mentioned notion of 'firability." In brief, an *interpretation* I assigns to each ground atom, an element of $\mathcal{T}$. Intuitively, if $\mathcal{T} = [0,1]$, then the assignment of 0.7 to atom A means that according to interpretation I, A is true with certainty 70% or more. Interpretation

I *satisfies* a ground annotated atom $A:\mu$ iff $\mu \preceq I(A)$. The notion of satisfaction of formulas containing other connectives, such as $\&, \vee, \leftarrow$ and quantifiers $\forall, \exists$ is the usual one [30]. In particular, I satisfies the ground annotated clause $A_0:\mu_0 \leftarrow (A_1:\mu_1 \,\&\ldots\&\, A_n:\mu_n)$ iff either $I \not\models (A_1:\mu_1 \,\&\ldots\&\, A_n:\mu_n)$ or $I \models A_0:\mu_0$. The symbol "$\models$" is read "satisfies." I satisfies a non-ground clause iff I satisfies each and every ground instance of the clause (with annotation variables instantiated to members of $\mathcal{T}$ and logical variables instantiated to logical terms).

2.2 Overview of Amalgamation Thoery

Suppose we have a collection of "local" databases $DB_1, \ldots, DB_n$ over a complete lattice $\mathcal{T}$ of truth values. In this section, we recall, from [32], how the theory of `GAP`s may be successfully applied to defining a *new* lattice of truth values that forms the basis for a "mediator" or "supervisory database." To do so, we first define the `DNAME` lattice; this is the power set, $2^{\{1,\ldots,n,\mathbf{s}\}}$. The integer i refers to database DB_i, while $\mathbf{s}$ refers to the supervisor. Note, in particular, that $2^{\{1,\ldots,n,\mathbf{s}\}}$ is a complete lattice under the set inclusion ordering.

We assume that we have a set DV of variables (called `DNAME` variables) ranging over $2^{\{1,\ldots,n,\mathbf{s}\}}$. If $A:\mu$ is an atom over lattice $\mathcal{T}$, V is a DNAME-variable, and $D \subseteq \{1,\ldots,n,\mathbf{s}\}$, then $A:[D,\mu]$ and $A:[V,\mu]$ are called *amalgamated atoms*. Intuitively, if $\mathcal{T} = [0,1]$, the amalgamated atom $\mathit{at\_robot}(3,4):[\{1,2,3\},0.8]$ says that according to the (joint) information of databases $1,2$ and 3, the degree of certainty that the robot is at location $(3,4)$ is 80% or more .

An amalgamated clause is a statement of the form:

$$A_0:[D_0,\mu_0] \quad \leftarrow \quad A_1:[D_1,\mu_1] \,\&\ldots\&\, A_n:[D_n,\mu_n]$$

where $A_0:[D_0,\mu_0],\ldots,A_n:[D_n,\mu_n]$ are amalgamated atoms. An *amalgamated database* is a collection of clauses of this form.

Mediator/Supervisory Database: Suppose $DB_1,\ldots,DB_n$ are GAPs. A *supervisory database*[1] S is a set of amalgamated clauses such that every ground instance of a clause in S is of the form:

$$A_0:[\{\mathbf{s}\},\mu] \quad \leftarrow \quad A_1:[D_1,\mu_1] \,\&\ldots\&\, A_n:[D_n,\mu_n]$$

where, for all $1 \leq i \leq (n+m)$, $D_i \subseteq \{1,\ldots,n,\mathbf{s}\}$.

Intuitively, ground instances of clauses in the supervisor say: "If the databases in set D_i, $1 \leq i \leq n$, (jointly) imply that the truth value of A_i is at least μ_i, then the supervisor will conclude that the truth value of A_0 is at least μ." This mode of expressing supervisory information is very rich – in [32], it is shown that we can express prioritized knowledge about predicates, prioritized knowledge about objects, as well as methods to achieve consensus.

We now define the concept of an *amalgam* of local databases $DB_1,\ldots,DB_n$ via a *supervisor* S. First, each clause C in DB_i of the form

$$A_0:\mu_0 \quad \leftarrow \quad A_1:\mu_1 \,\&\ldots\&\, A_n:\mu_n$$

[1]When the databases being integrated are geographically dispersed across a network, it is common to distribute the mediator so that bottlenecks (e.g. due to network problems) do not have a devastating effect. In this paper, we will not study issues relating to implementing distributed mediators (though we are doing so in a separate, concurrent effort).

is replaced by the amalgamated clause, $AT(C)$:

$$A_0 : [\{i\}, \mu_0] \quad \leftarrow \quad A_1 : [\{i\}, \mu_1] \,\&\, \ldots \,\&\, A_n : [\{i\}, \mu_n].$$

We use $AT(DB_i)$ to denote the set $\{AT(C) \mid C \in DB_i\}$.

The *amalgam* of $DB_1, \ldots, DB_n$ via a *supervisor* S is the amalgamated knowledge base $(S \cup \bigcup_{i=1}^{n} AT(DB_i))$.

The model theory for amalgamated knowledge bases is (slightly) different from that of individual GAPs because it must account for a new type of variable, viz. the DNAME variables. An $\mathbf{A} - interpretation$ J for an amalgamated database is a mapping from the set of ground atoms of our base language to the set of functions

$$\text{from } \{1, \ldots, n, \mathbf{s}\} \text{ to } \mathcal{T},$$

i.e. for $A \in B_L$, $I(A)$ is a mapping from $\{1, \ldots, n, \mathbf{s}\}$ to $\mathcal{T}$. In other words, if $I(A)(i) = \mu$, then according to the interpretation I, DB_i says the truth value of A is at least μ. Given a subset, D, of $\{1, \ldots, n, \mathbf{s}\}$ we use $I(A)(D)$ to denote $\sqcup_{i \in D}(J(A))(i)$. An $\mathbf{A} - interpretation$, J, satisfies the ground amalgamated atom $A : [D, \mu]$ iff $\mu \preceq \sqcup_{i \in D} (J(A))(i)$. Here, $\sqcup$ denotes "least upper bound (lub)". All the other symbols are interpreted in the same way as for ordinary $\mathcal{T}$-valued interpretations with the caveat that for quantification, DNAME variables are instantiated to subsets of $\{1, \ldots, n, \mathbf{s}\}$ and other annotation variables are instantiated to members of $\mathcal{T}$. Note that we will always use the word $\mathbf{A} - interpretation$ to denote an interpretation of an amalgamated KB and use the expression "interpretation" or "$\mathcal{T}$-interpretation" to refer to an interpretation of a GAP.

Linking Local and Amalgamated Database. We now show how we can go from models of amalgamated KBs to models of GAPs and vice-versa by using a concept of "projection" and "locale generation." We then prove a theorem that exhibits the strong links between models of the GAPs $DB_1, \ldots, DB_n$ and the amalgam of databases $DB_1, \ldots, DB_n$ via supervisor S.

Suppose Q is the amalgam of $(DB_1, \ldots, DB_n, S)$ and J is an $\mathbf{A}$-interpretation. The *projection of J on DB_i for $1 \leq i \leq n$* is the interpretation I defined as follows: $I(A) = J(A)(i)$.

While projections allow us to obtain an interpretation from an $\mathbf{A}$-interpretation, we may also need to obtain $\mathbf{A}$-interpretations from interpretations. Given an interpretation, I, the *locale* of I w.r.t. the *locale* of I w.r.t. a GAP DB_i, is the set $\{I' \mid I' \text{ is an } \mathbf{A}\text{-interpretation and for all ground atoms } A, I'(A)(i) = I(A)\}$.

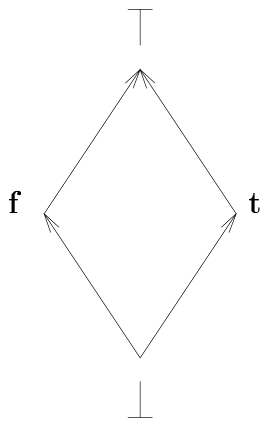
The following theorem now shows that the models of local databases (i.e. those interpretations that satisfy all facts and rules in the local database) are closely related to the $\mathbf{A}$-models of amalgams (i.e. the $\mathbf{A}$-interpretations that satisfy all the rules in the amalgam).

Amalgamation Theorem. ([32])

1. Suppose DB_i is a GAP and I is an interpretation such that there exists an I' in the locale of I which is an $\mathbf{A}$-model of $AT(DB_i)$. Then I is a model of DB_i.

2. $A : \mu$ is a logical consequence of DB_i (i.e. $A : \mu$ is satisfied by all interpretations that satisfy all clauses of DB_i) iff $A : [\{i\}, \mu]$ is an $\mathbf{A}$-consequence of $AT(DB_i)$.

3. Let Q be the amalgam of databases $(DB_1, \ldots, DB_n, S)$, and let J be an $\mathbf{A}$-model of Q. Then: the projection, J_i of J on DB_i is a model of DB_i. □



3 Motivation

In this section, we will present two motivating examples – the first is a set of deductive databases expressed using FOUR-valued logic describing a static robotic domain (i.e. one where the world remains constant). The second example extends this to reason about a dynamically changing world, and thus incorporates both uncertainty and time. These examples will be used throughout the paper to illustrate various intuitions as they arise in the paper.

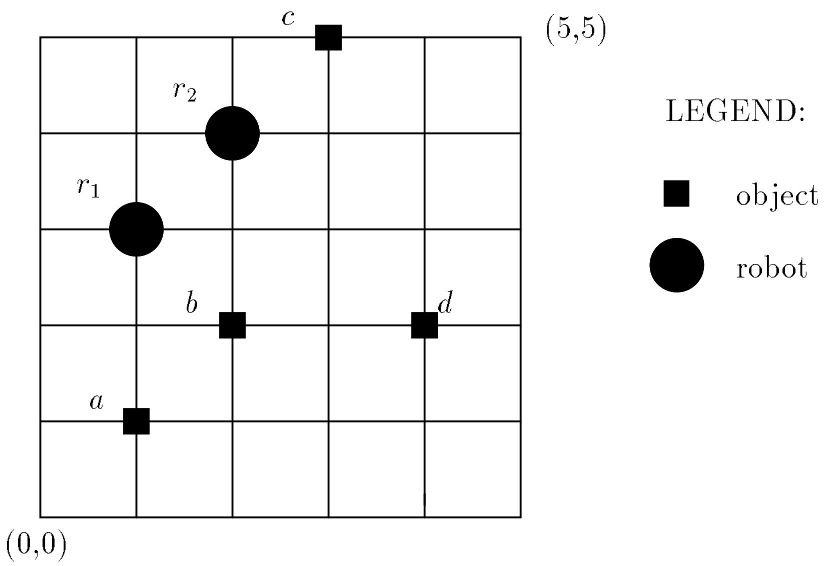
We will assume that the reader is familiar with generalized annotated programs (GAPs) as defined in [20].

3.1 Robot Example: Static Case

Consider two mobile robots, $r1$ and $r2$, that are operating in a common workspace. Each of these two robots has access to three databases; one of these databases represents information about the locations of objects in the workspace (cf. Figure 3.1), the second represents information about the weight of these objects, while the third represents information about the temperature of the objects. The last two databases also contain information about what kinds of loads the individual robots can lift. Each of these three databases is expressed over the lattice FOUR shown in Figure 3.1 and is shown below:

$\boxed{DB_1:}$

$$
\begin{array}{rcl}
at(r1,3,2):\mathbf{t} & \leftarrow & \\
at(r2,4,4):\mathbf{t} & \leftarrow & \\
at(a,1,1):\mathbf{t} & \leftarrow & \\
at(b,2,2):\mathbf{t} & \leftarrow & \\
at(c,3,5):\mathbf{t} & \leftarrow & \\
at(d,4,2):\mathbf{t} & \leftarrow & \\
right(E1,E2):\mathbf{t} & \leftarrow & at(E1,X1,Y1):\mathbf{t} \,\&\, at(E2,X2,Y1):\mathbf{t} \,\&\, X1 > X2. \\
left(E1,E2):\mathbf{t} & \leftarrow & at(E1,X1,Y1):\mathbf{t} \,\&\, at(E2,X2,Y1):\mathbf{t} \,\&\, X1 < X2. \\
above(E1,E2):\mathbf{t} & \leftarrow & at(E1,X1,Y1):\mathbf{t} \,\&\, at(E2,X1,Y2):\mathbf{t} \,\&\, Y1 > Y2.
\end{array}
$$



$$
\begin{aligned}
below(E1,E2):\mathbf{t} &\leftarrow at(E1,X1,Y1):\mathbf{t}\,\&\,at(E2,X1,Y2):\mathbf{t}\,\&\,Y1<Y2.\\
at(E1,X,Y):\mathbf{f} &\leftarrow at(E2,X,Y):\mathbf{t}\,\&\,E1\neq E2.
\end{aligned}
$$

This database specifies where the objects are located (including the robots), and also specifies relations such as "entity $E1$ is to the right of entity $E2$ if ... ," "entity $E1$ is to the left of $E2$ if ... ," "entity $E1$ is above entity $E2$ if ... " and "entity $E1$ is below entity $E2$ if ...." There is also a rule saying that two things cannot be at the same place. We assume that relations like $>$, $<$, and $=$ are evaluated in the standard way. Intuitively, the first rule above says "If the entity $E1$ is at location $(X1,Y2)$ and entity $E2$ is at location $(X2,Y2)$ and $X1>X2$, then $E1$ is to the right of $E2$." When the annotation $\mathbf{t}$ associated with an atom A leads to the reading: "A's truth value is at least $\mathbf{t}$." In general, $\mathbf{t}$ may be replaced with a different annotation, μ, picked from a complete lattice of truth values and the same reading holds with "at least" referring to the lattice ordering.

$\boxed{DB_2:}$

$$
\begin{aligned}
weight(a,36):\mathbf{t} &\leftarrow\\
weight(b,19):\mathbf{t} &\leftarrow\\
weight(c,48):\mathbf{t} &\leftarrow\\
weight(d,27):\mathbf{t} &\leftarrow\\
can\_lift(r1,X):\mathbf{t} &\leftarrow weight(X,W):\mathbf{t}\,\&\,W<50.\\
can\_lift(r1,X):\mathbf{f} &\leftarrow weight(X,W):\mathbf{t}\,\&\,W\geq 50.\\
can\_lift(r2,X):\mathbf{t} &\leftarrow weight(X,W):\mathbf{t}\,\&\,W<30.\\
can\_lift(r2,X):\mathbf{f} &\leftarrow weight(X,W):\mathbf{t}\,\&\,W\geq 30.
\end{aligned}
$$

$\boxed{DB_3:}$

$$
\begin{aligned}
temp(a,92):\mathbf{t} &\leftarrow \\
temp(b,61):\mathbf{t} &\leftarrow \\
temp(c,55):\mathbf{t} &\leftarrow \\
temp(d,112):\mathbf{t} &\leftarrow \\
can\_lift(r1,X):\mathbf{t} &\leftarrow temp(X,T):\mathbf{t} \,\&\, T<60. \\
can\_lift(r1,X):\mathbf{f} &\leftarrow temp(X,T):\mathbf{t} \,\&\, T\geq 60. \\
can\_lift(r2,X):\mathbf{t} &\leftarrow temp(X,T):\mathbf{t} \,\&\, T<120. \\
can\_lift(r2,X):\mathbf{f} &\leftarrow temp(X,T):\mathbf{t} \,\&\, T\geq 120.
\end{aligned}
$$

Using DB_2 alone, we may conclude that $r1$ can lift any of a,b,c,d, while using DB_3 alone, we may conclude that $r1$ can lift only c. Similarly, DB_2 alone tells us that $r2$ can lift b and d, while using DB_3 alone, we may conclude that $r2$ can lift all of a,b,c and d. Clearly this leads to inconsistency. In addition to resolving such conflicts, we may wish to coordinate what should be done by the two robots $r1$ and $r2$. A supervisory database is a database that specifies how to resolve such conflicts and how to achieve the desired coordination. For instance it may be the case that $r1$ moves easily in the vertical direction, while $r2$ moves easily in the horizontal direction. If an object is above or below $r1$, and the supervisor determines that $r1$ can lift that object, then the supervisor may decide to command $r1$ to lift that object. Similarly, if an object is to the left or right of $r2$, and the supervisor determines that $r2$ can lift that object, then the supervisor may decide to command $r2$ to lift that object. If the object is not exactly above or below $r1$ or to the right, left of $r2$, then the supervisor will first command $r1$ to lift the object. If no command is issued to $r1$ to lift an object, then $r2$ will be commanded to lift that object. These are formalized using the following "supervisory" knowledge base.

$$
\begin{aligned}
can\_lift(r1,X):[\{\mathbf{s}\},V] &\leftarrow can\_lift(r1,X):[\{2,3\},V]. \\
can\_lift(r2,X):[\{\mathbf{s}\},V_1\sqcap V_2] &\leftarrow can\_lift(r2,X):[\{2\},V_1] \,\&\, can\_lift(r2,X):[\{3\},V_2]. \\
command\_lift(X,r1):[\{\mathbf{s}\},V] &\leftarrow can\_lift(r1,X):[\{\mathbf{s}\},V] \,\&\, above(X,r1):[\{1\},\mathbf{t}]. \\
command\_lift(X,r1):[\{\mathbf{s}\},V] &\leftarrow can\_lift(r1,X):[\{\mathbf{s}\},V] \,\&\, below(X,r1):[\{1\},\mathbf{t}]. \\
command\_lift(X,r2):[\{\mathbf{s}\},V] &\leftarrow can\_lift(r2,X):[\{\mathbf{s}\},V] \,\&\, left(X,r2):[\{1\},\mathbf{t}]. \\
command\_lift(X,r2):[\{\mathbf{s}\},V] &\leftarrow can\_lift(r2,X):[\{\mathbf{s}\},V] \,\&\, right(X,r2):[\{1\},\mathbf{t}]. \\
command\_lift(X,r1):[\{\mathbf{s}\},V] &\leftarrow can\_lift(r1,X):[\{\mathbf{s}\},V]. \\
command\_lift(X,r2):[\{\mathbf{s}\},\mathbf{t}] &\leftarrow can\_lift(r2,X):[\{2,3\},\mathbf{t}] \,\&\, command\_lift(X,r1):[\{\mathbf{s}\},\mathbf{f}].
\end{aligned}
$$

The first two rules in the above supervisory knowledge base are very interesting. As far as robot $r1$ is concerned, the supervisor is willing to accept the truth value provided by any of the databases – in other words, the supervisor is indecisive and acts as if both what DB_2 says is correct and what DB_3 says is correct (even though they may contradict each other). This may be an appropriate strategy when robot $r1$ is a very inexpensive robot, and the task of lifting the objects is critical. The second rule says that the supervisor only concludes that $r2$ can lift an object if both databases DB_2 and DB_3 say it can (consensus).

The amalgam of local databases DB_1, DB_2, DB_3 with the supervisory database S, as defined in [32] is:

$$
\begin{array}{rcl}
at(r1,3,2):[\{1\},\mathbf{t}] & \leftarrow & \\
at(r2,4,4):[\{1\},\mathbf{t}] & \leftarrow & \\
at(a,1,1):[\{1\},\mathbf{t}] & \leftarrow & \\
at(b,2,2):[\{1\},\mathbf{t}] & \leftarrow & \\
at(c,3,5):[\{1\},\mathbf{t}] & \leftarrow & \\
at(d,4,2):[\{1\},\mathbf{t}] & \leftarrow & \\
right(E1,E2):[\{1\},\mathbf{t}] & \leftarrow & at(E1,X1,Y1):[\{1\},\mathbf{t}]\,\&\,at(E2,X2,Y1):[\{1\},\mathbf{t}]\,\&\,X1>X2. \\
left(E1,E2):[\{1\},\mathbf{t}] & \leftarrow & at(E1,X1,Y1):[\{1\},\mathbf{t}]\,\&\,at(E2,X2,Y1):[\{1\},\mathbf{t}]\,\&\,X1<X2. \\
above(E1,E2):[\{1\},\mathbf{t}] & \leftarrow & at(E1,X1,Y1):[\{1\},\mathbf{t}]\,\&\,at(E2,X1,Y2):[\{1\},\mathbf{t}]\,\&\,Y1>Y2. \\
below(E1,E2):[\{1\},\mathbf{t}] & \leftarrow & at(E1,X1,Y1):[\{1\},\mathbf{t}]\,\&\,at(E2,X1,Y2):[\{1\},\mathbf{t}]\,\&\,Y1<Y2. \\
at(E1,X,Y):[\{1\},\mathbf{f}] & \leftarrow & at(E2,X,Y):[\{1\},\mathbf{t}]\,\&\,E1\neq E2. \\
weight(a,36):[\{2\},\mathbf{t}] & \leftarrow & \\
weight(b,19):[\{2\},\mathbf{t}] & \leftarrow & \\
weight(c,48):[\{2\},\mathbf{t}] & \leftarrow & \\
weight(d,27):[\{2\},\mathbf{t}] & \leftarrow & \\
can\_lift(r1,X):[\{2\},\mathbf{t}] & \leftarrow & weight(X,W):[\{2\},\mathbf{t}]\,\&\,W<50. \\
can\_lift(r1,X):[\{2\},\mathbf{f}] & \leftarrow & weight(X,W):[\{2\},\mathbf{t}]\,\&\,W\geq 50. \\
can\_lift(r2,X):[\{2\},\mathbf{t}] & \leftarrow & weight(X,W):[\{2\},\mathbf{t}]\,\&\,W<30. \\
can\_lift(r2,X):[\{2\},\mathbf{f}] & \leftarrow & weight(X,W):[\{2\},\mathbf{t}]\,\&\,W\geq 30. \\
temp(a,92):[\{3\},\mathbf{t}] & \leftarrow & \\
temp(b,61):[\{3\},\mathbf{t}] & \leftarrow & \\
temp(c,55):[\{3\},\mathbf{t}] & \leftarrow & \\
temp(d,112):[\{3\},\mathbf{t}] & \leftarrow & \\
can\_lift(r1,X):[\{3\},\mathbf{t}] & \leftarrow & temp(X,T):[\{3\},\mathbf{t}]\,\&\,T<60. \\
can\_lift(r1,X):[\{3\},\mathbf{f}] & \leftarrow & temp(X,T):[\{3\},\mathbf{t}]\,\&\,T\geq 60. \\
can\_lift(r2,X):[\{3\},\mathbf{t}] & \leftarrow & temp(X,T):[\{3\},\mathbf{t}]\,\&\,T<120. \\
can\_lift(r2,X):[\{3\},\mathbf{f}] & \leftarrow & temp(X,T):[\{3\},\mathbf{t}]\,\&\,T\geq 120. \\
can\_lift(r1,X):[\{\mathbf{s}\},V] & \leftarrow & can\_lift(r1,X):[\{2,3\},V.] \\
can\_lift(r2,X):[\{\mathbf{s}\},V_1\sqcap V_2] & \leftarrow & can\_lift(r2,X):[\{2\},V_1]\,\&\,can\_lift(r2,X):[\{3\},V_2]. \\
command\_lift(X,r1):[\{\mathbf{s}\},V] & \leftarrow & can\_lift(r1,X):[\{\mathbf{s}\},V]\&above(X,r1):[\{1\},\mathbf{t}]. \\
command\_lift(X,r1):[\{\mathbf{s}\},V] & \leftarrow & can\_lift(r1,X):[\{\mathbf{s}\},V]\&below(X,r1):[\{1\},\mathbf{t}]. \\
command\_lift(X,r2):[\{\mathbf{s}\},V] & \leftarrow & can\_lift(r2,X):[\{\mathbf{s}\},V]\&left(X,r2):[\{1\},\mathbf{t}]. \\
command\_lift(X,r2):[\{\mathbf{s}\},V] & \leftarrow & can\_lift(r2,X):[\{\mathbf{s}\},V]\&right(X,r2):[\{1\},\mathbf{t}]. \\
command\_lift(X,r1):[\{\mathbf{s}\},V] & \leftarrow & can\_lift(r1,X):[\{\mathbf{s}\},V]. \\
command\_lift(X,r2):[\{\mathbf{s}\},\mathbf{t}] & \leftarrow & can\_lift(r2,X):[\{2,3\},\mathbf{t}]\&command\_lift(X,r1):[\{\mathbf{s}\},\mathbf{f}].
\end{array}
$$

3.2 Robot Example: Dynamic Case:

In the preceding section, we have not taken into account, the fact that the workspace may be changing with time – in other words, robot $r1$ may need to base its actions on its perceptions of what robot $r2$ may do (even in the future). For instance, suppose we assume that all changes in the workspace occur at discrete time intervals, and that at time 0, the workspace is as shown in Figure 3.1. Let us suppose that the following events occur:

- At time 1, robot $r1$ moves "right" one location, i.e. moves to location $(2, 3)$.
- At time 2, robot $r1$ again moves right one location, i.e. it moves to location $(3, 3)$.

At this point, robot $r2$ may well conclude, based on its knowledge of robot $r1$'s past actions, that $r1$ will, in all likelihood [2], continue in its rightward path. It may formalize this intuition as a clause that says: "Given that robot $r1$ moves right at time T and $T+1$ with probabilities V_1, V_2 respectively, then the probability that it will continue to move right is a function, f, is some function of V_1, V_2." Using the truth value lattice $[0,1] \times 2^{\mathbf{R}^+}$, we may encode this information as:

$$\begin{array}{rcl} at(r1, X+2, Y) : [f(V_1, V_2), \{T+2\}] & \leftarrow & at(r1, X+1, Y) : [V_1, \{T+1\}]\ \& \\ & & at(r1, X, Y) : [V_2, \{T\}]. \\ at(r1, X+1, Y+1) : [1 - f(V_1, V_2), \{T+2\}] & \leftarrow & at(r1, X+1, Y) : [V_1, \{T+1\}]\ \& \\ & & at(r1, X, Y) : [V_2, \{T\}]. \end{array}$$

Here, T is a variable ranging over time points, and V, V_1, V_2 are variables ranging over the unit interval $[0, 1]$. The first clause says that if $r1$ is at location (X, Y) at time T with certainty V_2, and $r1$ is at location $(X+1, Y)$ at time $(T+1)$ with certainty V_1, then the certainty of its being at location $(X+2, Y)$ at time $(T+2)$ is $f(V_1, V_2)$ where f is some function from $[0,1] \times [0,1]$ to $[0,1]$. The second clause says that the probability that the robot will be at location $(X+1, Y+1)$ at time $(T+2)$ (i.e. it moves "down" instead of "right") is $1 - f(V_1, V_2)$.

The truth value lattice being used in this example is $[0,1] \times 2^{\mathbf{R}^+}$. An annotated atom of the form $A : [u, t]$ intuitively says that "for all time points $t^* \in t$, atom A is true with certainty u or more." The lattice ordering on $[0,1] \times 2^{\mathbf{R}^+}$ is defined as: $[u_1, t_1] \leq [u_2, t_2]$ iff $u_1 \leq u_2$ and $t_1 \subseteq t_2$.

The detailed description of annotated logics is beyond the scope of this paper – it is well-documented in the literature [7, 20, 32, 33].

4 A Resolution-Based Query Processing Procedure

In this section, we will develop a framework for processing queries to amalgamated databases. This procedure is a resolution-based procedure, and hence, inherits many of the disadvantages of existing resolution-based strategies. It is similar to work by Lu, Murray and Rosenthal [24] who have independently developed a more general framework for query processing in GAPs. The work described

[2] This is only an illustration. In the real-world, a robot may use far more complex strategies to make predictions on what other agents in the workspace may do.

here is intended as a stepping stone for the development of a more sophisticated procedure, called MULTI_OLDT, that will be described in Section 5.

We now define the concept of the *up-set* of an annotation, or a set of annotations. Intuitively, given a set Q of annotations, the up-set of Q is simply the set of all elements in the truth value lattice that are larger than some element in Q.

Definition 1 Suppose $\langle \mathcal{R}; \leq \rangle$ is a partially ordered set and $Q \subseteq \mathcal{R}$. Then, $\Uparrow Q = \{y \in \mathcal{R} \mid (\exists x \in Q) x \leq y\}$ and $\Downarrow Q = \{y \in \mathcal{R} \mid (\exists x \in Q) y \leq x\}$.

For instance, if we consider the lattice FOUR described earlier, it turns out that $\Uparrow \mathbf{t} = \{\mathbf{t}, \top\}$. Similarly, $\Uparrow \{\mathbf{t}, \mathbf{f}\} = \{\mathbf{t}, \mathbf{f}, \top\}$.

Up-sets may be used to capture the following intuition: suppose we consider an amalgamated atom $A : [D, \mu]$. Then this atom is satisfied by any **A**-interpretation I such that $\mu \preceq \sqcup_{d \in D} I(A)(d)$. In other words, satisfaction of $A : [D, \mu]$ by I requires that $\sqcup_{d \in D} I(A)(d) \in \Uparrow \mu$. This leads us to consider the possibility of extending amalgamated atoms to have a *set of truth values* as the second element of its annotation.

Definition 2 Given an amalgamated annotation $[D, \mu]$ where $D \subseteq \mathcal{V} = \{1, .., n, \mathbf{s}\}$ and $\mu \in \mathcal{T}$, and a function $f_s : (2^{\mathcal{V}} \times \mathcal{T}) \rightarrow (2^{\mathcal{V}} \times 2^{\mathcal{T}})$ the expression $f_s([D, \mu])$ is called a *set expansion* of $[D, \mu]$.

For example, we may take f_s to be the function such that $f_s([D, \mu]) = [D, \Uparrow \mu]$, or we may take f_s to be the function such that $f_s([D, \mu]) = [D, \mathcal{T} \setminus \Uparrow \mu]$. If we take f_s to be the latter, and we consider the lattice FOUR, then $f_s([D, \mathbf{t}] = [D, \{\perp, \mathbf{f}\}]$.

It will turn out that the two examples of f_s given above will be particularly important – hence, we give special names to these functions below.

Definition 3 Given an amalgamated annotation $[D, \mu]$ where $D \subseteq \mathcal{V} = \{1, .., n, s\}$ and $\mu \in \mathcal{T}$, the regular set expansion, EXP, is given by EXP$([D, \mu]) = [D, \Uparrow \mu]$. Similarly, the complement of the regular expansion is given by COMP$([D, \mu]) = [D, \mathcal{T} \setminus \Uparrow \mu]$.

Example 1 Let $\mathcal{T} = \{\top, \mathbf{t}, \mathbf{f}, \perp\}$ and $\mathcal{V} = \{1, 2, \mathbf{s}\}$.

$$\texttt{EXP}([\{1,2\}, \mathbf{t}]) = [\{1,2\}, \{\mathbf{t}, \top\}]$$

$$\texttt{COMP}([\{1,2\}, \mathbf{t}]) = [\{1,2\}, \{\mathbf{f}, \perp\}]$$

□

In the sequel, we will often use the notation μ_s to denote a *set of truth values (annotations)*. Thus, $A : [D, \mu_s]$ is intuitively read as: "The truth value of A, as determined jointly by the databases in D is in the set μ_s." The following definition defines an asymmetric notion of of "intersection" of two amalgamated annotations.

Definition 4 Given two set expansions $[D_1, \mu_{s_1}]$,$[D_2, \mu_{s_2}]$ where $D_1, D_2 \subseteq \{1, \ldots, n, \mathbf{s}\}$ and $\mu_{s_1}, \mu_{s_2} \in 2^{\mathcal{T}}$, the partial function S-INT is defined as follows:

$$\text{S-INT}([D_1, \mu_{s_1}], [D_2, \mu_{s_2}]) = \begin{cases} [D_1, \mu_{s_1} \cap \mu_{s_2}], & \text{if } D_2 \subseteq D_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that S-INT is asymmetric because of the subset condition. This asymmetry is a key distinction between our work and the concurrently developed work of [24].

For example, suppose we consider the lattice `FOUR` and consider the amalgamated annotations: $[\{1,2,3\}, \{\mathbf{t}, \top\}]$ and $[\{1,3\}, \{\mathbf{f}, \top\}]$, then

$$\text{S-INT}([\{1,2,3\}, \{\mathbf{t}, \top\}], [\{1,3\}, \{\mathbf{f}, \top\}]) \quad = \quad [\{1,2,3\}, \{\top\}]$$

but $\text{S-INT}([\{1,3\}, \{\mathbf{f}, \top\}], [\{1,2,3\}, \{\mathbf{t}, \top\}])$ is undefined.

Using the concept of set expansions of amalgamated atoms, we now define the concept of a *regular representation* of a clause. Later in this section, we will define a resolution-based strategy that uses regular representations of amalgamated clauses instead of the amalgamated clauses themselves. The advantage is that the expensive *reductant* rule of inference introduced by Kifer and Lozinskii [19] and later studied by Kifer and Subrahmanian [20] can be eliminated by using regular representations.

Definition 5 Given a clause C of the form:

$$A_0 : [D_0, \mu_0] \leftarrow A_1 : [D_1, \mu_1] \& \ldots \& A_n : [D_n, \mu_n]$$

the *regular representation* of C, denoted by C^*, is the expression:

$$A_0 : \mathtt{EXP}([D_0, \mu_0]) \leftarrow A_1 : \mathtt{EXP}([D_1, \mu_1]) \& \ldots \& A_n : \mathtt{EXP}([D_n, \mu_n])$$

In other words the regular representation is obtained by set expanding all the amalgamated annotations using the expansion function `EXP`.

Example 2 (Static Robot Example Revisited) Consider the following rule from DB_2 of the Static Robot example.

$$can\_lift(r1, X) : \mathbf{t} \quad \leftarrow \quad weight(X, W) : \mathbf{t} \,\&\, W < 50.$$

The amalgamated form of this, as defined in [32], is

$$can\_lift(r1, X) : [\{2\}, \mathbf{t}] \quad \leftarrow \quad weight(X, W) : [\{2\}, \mathbf{t}] \,\&\, W < 50.$$

The regular representation of this is:

$$can\_lift(r1, X) : \mathtt{EXP}([\{2\}, \mathbf{t}]) \quad \leftarrow \quad weight(X, W) : \mathtt{EXP}([\{2\}, \mathbf{t}]) \,\&\, W < 50.$$

(We assume that the constraint $W < 50$ is a predefined evaluable relation). □

Example 3 (Dynamic Robot Example Revisited) Suppose the rule below occurs in database i of the Dynamic Robot example:

$$\begin{array}{rcl} at(r1, X+2, Y) : [f(V_1, V_2), \{T+2\}] & \leftarrow & at(r1, X+1, Y) : [V_1, \{T+1\}]\,\& \\ & & at(r1, X, Y) : [V_2, \{T\}]. \end{array}$$

Then the regular representation of this clause is:

$$\begin{array}{rcl} at(r1, X+2, Y) : \mathtt{EXP}([\{i\}, [f(V_1, V_2), \{T+2\}]]) & \leftarrow & at(r1, X+1, Y) : \mathtt{EXP}([\{i\}, [V_1, \{T+1\}]])\,\& \\ & & at(r1, X, Y) : \mathtt{EXP}([\{i\}, [V_2, \{T\}]]). \end{array}$$

□

Definition 6 (S-satisfaction) An **A**–interpretation I S-satisfies an expanded atom $A : [D, \mu_s]$ where $D \subseteq \{1, \ldots, n, \mathbf{s}\}$ and $\mu_s \in 2^{\mathcal{T}}$ iff $I \models^{\mathbf{A}} A : [D, \mu]$ for some $\mu \in \mu_s$.

The notion of an S-logical consequence is similar to that in classical logic – only now, S-satisfaction is considered instead of ordinary satisfaction.

Definition 7 An amalgamated atom in set expansion form $A : f_{s_1}([D_1, \mu_1])$ is said to be an S-consequence of another atom $B : f_{s_2}[D_2, \mu_2]$ (denoted by $B : f_{s_2}([D_2, \mu_2]) \models^{\mathrm{S}} A : f_{s_1}[D_1, \mu_1]$), iff any **A**-*interpretation* I that S-satisfies $B : f_{s_2}([D_2, \mu_2])$ also S-satisfies $A : f_{s_1}([D_1, \mu_1])$.

Example 4 Let the truth value lattice be $\mathtt{FOUR}$ and let I be an **A**-interpretation such that $I(A)(1) = \bot$ and $I(A)(2) = \mathbf{t}$. $\bigsqcup_{d \in \{1,2\}} I(A)(d) = \mathbf{t}$. Hence, I S-satisfies $A : [\{1, 2\}, \{\mathbf{t}, \mathbf{f}, \top\}]$ since $t \in \{\mathbf{t}, \mathbf{f}, \top\}$. □

Just as we defined the notion of "regular representation" of clauses, we also need to define the notion of "regular representation" of queries.

Definition 8 A *query* Q is a statement of the form:

$$\leftarrow A_1 : [D_1, \mu_1] \& \ldots \& A_n : [D_m, \mu_m]$$

where all the free variables of the query are assumed to be universally quantified[3]. The regular representation of the query Q, denoted Q^* is the query:

$$A_1 : \mathtt{COMP}([D_1, \mu_1]) \vee \ldots \vee A_n : \mathtt{COMP}([D_m, \mu_m]) \leftarrow$$

The following result follows immediately from the definitions and is given without proof.

Proposition 1 Suppose I is an **A**-interpretation.

1. I satisfies a ground clause C iff I S-satisfies C^*.
2. I satisfies a ground query Q iff I S-satisfies Q^*.

□

We now come to the central concept in this section, viz. that of an S-resolvent.

[3] A query can be thought of as a headless Horn-clause. The negation of the above query is the statement $(\exists)(A_1 : [D_1, \mu_1] \& \ldots \& A_n : [D_m, \mu_m])$.

Definition 9 (S-resolution) Let C^* be the regular representation of a clause C and be given by:

$$A_0 : \mathtt{EXP}([D_0, \mu_0]) \leftarrow A_1 : \mathtt{EXP}([D_1, \mu_1]) \& \ldots \& A_n : \mathtt{EXP}([D_n, \mu_n])$$

and let W be the expression

$$B_1 : [D_{q_1}, \mu_{q_{s_1}}] \vee \ldots \vee B_m : [D_{q_m}, \mu_{q_{s_m}}] \leftarrow$$

where $[D_{q_i}, \mu_{q_{s_i}}], 1 \leq j \leq m$, are in set expansion form. Suppose B_i and A_0 are unifiable via mgu θ and suppose $D_0 \subseteq D_{q_i}$. Then the S-resolvent of W and C^* is the expression:

$$\begin{array}{ll} (& A_1 : \mathtt{COMP}([D_1, \mu_1]) \vee \ldots \vee A_n : \mathtt{COMP}([D_n, \mu_n]) \vee \\ & B_1 : [D_{q_1}, \mu_{q_{s_1}}] \vee \ldots \vee B_{i-1} : [D_{q_{i-1}}, \mu_{q_{s_{i-1}}}] \vee B_{i+1} : [D_{q_{i+1}}, \mu_{q_{s_{i+1}}}] \vee \ldots \vee B_m : [D_{q_m}, \mu_{q_{s_m}}] \vee \\ & A_0 : \text{S-INT}([D_{q_i}, \mu_{q_{s_i}}], \mathtt{EXP}([D_0, \mu_0]))\ )\theta \ \leftarrow \end{array}$$

In case, S-INT $([D_{q_i}, \mu_{q_{s_i}}], \mathtt{EXP}([D_0, \mu_0])) = [D, \mu_s]$ is ground and μ_s evaluates to $\emptyset$, then we simplify the above S-resolvent by removing the atom $\left(A_0 : \text{S-INT}\left([D_{q_i}, \mu_{q_{s_i}}], \mathtt{EXP}\left([D_0, \mu_0]\right)\right)\right)\theta$.

Two important points that distinguish S-resolution for amalgamated knowledge bases from GAPs are the following:

- First, it is possible that no atom may be "eliminated" during an S-resolution step. This occurs if μ_s above is not equal to $\emptyset$.
- Second, S-resolvents are inherently asymmetric because they are defined in terms of the S-INT operator which is not symmetric.

Before proceeding to study soundness and completeness issues pertaining to S-resolution, we present an example.

Example 5 Consider the truth value lattice FOUR. Let C be the clause

$$p(a) : [\{1\}, \{\top\}] \leftarrow$$

and let Q^* be the (regular representation)

$$p(X) : [\{1, 2\}, \{\mathbf{f}, \perp\}] \leftarrow .^4$$

$\theta = \{X = a\}$ is the mgu of $p(a)$ and $p(X)$, and hence C and Q can be S-resolved, yielding

$$(p(X) : [\{1, 2\}, \{\mathbf{f}, \perp\} \cap \{\top\}] \leftarrow)\{X = a\}$$

as the S-resolvent. This is reduced to the empty clause because $\{\mathbf{f}, \perp\} \cap \{\top\} = \emptyset$. $\square$

[4] Note that this query is the regular representation of

$$\leftarrow p(X) : [\{1, 2\}, \mathbf{t}]$$

Definition 10 An S-*deduction* from a query Q_0 and an amalgamated knowledge base AKB is a sequence:

$$\langle Q_0^*, C_0^*, \theta_0\rangle, \ldots, \langle Q_n^*, C_n^*, \theta_n\rangle$$

such that Q_{i+1}^* is an S-*resolvent* of Q_i^* and C_i^* via mgu θ_i, $(0 \leq i < n)$. Q_0^* is the regular representation of Q_0 and C_i^* is the regular representation of some clause C, $(0 \leq i \leq n)$.

An S-deduction is called an S-*refutation* if it is finite and the last query is the empty clause.

Lu, Murray and Rosenthal[24] have proved a more general version of the soundness and completeness result which we therefore state without proof[5].

Theorem 1 (Soundness of S-resolution) Suppose I S-satisfies a clause $C^* \equiv A_0 : \mathtt{EXP}([D_0, \mu_0]) \leftarrow A_1 : \mathtt{EXP}([D_1, \mu_1])$ &...& $A_n : \mathtt{EXP}([D_n, \mu_n])$ and a query $Q_k^* \equiv B_1 : [D_{q_1}, \mu_{q_{s_1}}] \vee \ldots \vee B_m : [D_{q_m}, \mu_{q_{s_m}}] \leftarrow$. Then, I S-satisfies the S-resolvent of C^* and Q_k^*. $\Box$

The following definition from [32] is needed for proving the Completeness results for amalgamated knowledge bases. Given an amalgamated knowledge base Q, it is possible to associate with Q, an operator $\mathbf{A}_Q$ that maps $\mathbf{A}$-interpretations to $\mathbf{A}$-interpretations.

Definition 11 [32] Suppose Q is an amalgamated knowledge base. We may associate with Q, an operator, $\mathbf{A}_Q$, that maps $\mathbf{A}$-interpretations to $\mathbf{A}$-interpretations as follows.

$\mathbf{A}_Q^{'}(I)(A)(D) = \sqcup\{\mu \mid A : [D, \mu] \leftarrow B_1 : [D_1, \mu_1] \& \ldots \& B_n : [D_n, \mu_n] \& \ \mathbf{not}(B_{n+1} : [D_{n+1}, \mu_{n+1}]) \& \ldots \& \mathbf{not}(B_{n+m} : [D_{n+m}, \mu_{n+m}])\}$ is a ground instance of a clause in Q and for all $1 \leq i \leq n$,$\mu_i \leq I(B_i)(D_i)$ and for all $(n+1) \leq j \leq (n+m)$,$\mu_j \not\leq I(B_j)(D_j)$.

$\mathbf{A}_Q(I)(A)(D) = \sqcup_{D' \subseteq D} \mathbf{A}_Q^{'}(I)(A)(D')$, for all $D \subseteq \{1, \ldots, n, s\}$.

Subrahmanian [32] proved that when Q is negation-free, $\mathbf{A}_Q$ is monotonic. Hence, $\mathbf{A}_Q$ has a least fixpoint which is identical to $\mathbf{A}_Q \uparrow \eta$ for some ordinal η. Unlike ordinary logic programs, even if η is ω, it is possible that $(\mathbf{A}_Q \uparrow \omega)(A)(i) = \mu$, but there is no integer $j < \omega$ such that $(\mathbf{A}_Q \uparrow j)(A)(i) = \mu$. This may occur because μ is the lub of an infinite sequence, $\mu_0, \mu_1, \ldots$ where $\mu_k = (\mathbf{A}_Q \uparrow k)(A)(i)$.

An amalgamated knowledge base is said to possess the fixpoint reachability property iff whenever $(\mathbf{A}_Q \uparrow \eta)(A)(i) = \mu$, there is an integer $j < \omega$ such that $(\mathbf{A}_Q \uparrow j)(A)(i) = \mu$. The fixpoint reachability property is critical for completeness because otherwise, we need to take recourse to infinitary proofs. It is well-known [20] that even in the case of GAPs, the fixpoint reachability property is critically necessary for obtaining completeness results. The proof of the following result is contained in Appendix A.

Theorem 2 (Completeness of S-resolution) Suppose $P \models Q$ where P is an amalgamated knowledge base that possesses the fixpoint reachability property. Then, there is an S-refutation of $(\leftarrow Q)^*$ from P. $\Box$

[5]NOTE TO THE REFEREES: The proof is included in Appendix A, but this appendix can be removed when the paper goes to press.

5 MULTI_OLDT Resolution

The previous section describes a sound and complete proof procedure for amalgamated knowledge bases. The completeness result for S-resolution asserts the existence of a refutation for $(\leftarrow Q)^*$ whenever Q is a logical consequence of a program P possessing the fixpoint reachability property.

However, the procedure does not:

- specify how to find a refutation, and
- does not specify how to handle queries which contain annotation variables.

The ability to specify, and process, queries such as "What is the (maximal) degree of certainty V that robot $r1$ will be at location $(4,3)$ at time instant 3 ?" is one that cannot be adequately handled by the "ground annotation" procedure described in Section 4. However, these are natural questions to ask – robot $r2$ may base its actions on the certainty with which it can conclude that robot $r1$ will be at a given location at a given time. In general, this problem can be characterized by the following maximization problem:

> *Given an atom A (whose truth value we want to find out) and a set D of local databases, find the maximal truth value V such that $A : [D, V]$ is an S-consequence of the amalgamated knowledge base P.*

Second, the robot may have a *hard deadline* within which to perform its action(s). Thus, it should have the ability to *interrupt* the query processing module and request the "best" answer obtained thus far.

How these two goals are achieved efficiently is the subject of this section of the paper. As a preview, we give a small example.

Example 6 Consider the databases DB_1, DB_2 and DB_3 in the static robot example, and suppose we ask the query:

$$\leftarrow can\_lift(r1, b) : [\{1, 2, 3\}, V].$$

The query Q says: "What is the maximal truth value V such that $can\_lift(r1, b) : [\{1, 2, 3\}, V]$ can be concluded ?" Q^* is: $can\_lift(r1, b) : [\{1, 2, 3\}, \mathcal{T} - \Uparrow V] \leftarrow$. Let us see what happens.

1. Resolving this query with the (regular representation of the) first rule in DB_2 yields, as resolvent, $Q_1^* \equiv$:

   $$can\_lift(r1, b) : [\{1, 2, 3\}, (\mathcal{T} - \Uparrow V) \cap \Uparrow \mathbf{t}] \vee weight(b, W) : [\{2\}, \mathcal{T} - \Uparrow \mathbf{t}] \vee W \geq 50 \leftarrow .$$

2. Resolving this query with the (regular representation of the) second fact in DB_2 yields

   $$can\_lift(r1, b) : [\{1, 2, 3\}, (\mathcal{T} - \Uparrow V) \cap \Uparrow \mathbf{t}] \vee weight(b, 19) : [\{2\}, (\mathcal{T} - \Uparrow \mathbf{t}) \cap \Uparrow \mathbf{t}] \vee 19 \geq 50 \leftarrow .$$

   As $(\mathcal{T} - \Uparrow \mathbf{t}) \cap \Uparrow \mathbf{t} = \emptyset$, the atom $weight(b, 19) : [(\mathcal{T} - \Uparrow \mathbf{t}) \cap \Uparrow \mathbf{t}]$ can be eliminated from the resolvent, and the evaluable atom $19 \geq 50$ may also be so eliminated, thus leaving us with the resolvent

   $$can\_lift(r1, b) : [\{1, 2, 3\}, (\mathcal{T} - \Uparrow V) \cap \Uparrow \mathbf{t}] \leftarrow .$$

   Note that at this stage, we are in a position to conclude that V must be at least $\mathbf{t}$ *for the following reasons:*

- All atoms in the body of the first rule in DB_2 have been resolved away (i.e. the subgoals generated by atoms in the body of this rule have been achieved), and
- $V = \mathbf{t}$ represents the *maximal* lattice value such that

$$(\mathcal{T} - \Uparrow V) \cap \Uparrow \mathbf{t} = \emptyset.$$

Hence, we may conclude that V's truth value is *at least* $\mathbf{t}$ (w.r.t. the lattice ordering).

3. After concluding that V's truth value is *at least* $\mathbf{t}$, we continue resolving the query from (2) above. We resolve it with the second clause in DB_3 to get:

$$can\_lift(r1, b) : [\{1,2,3\}, (\mathcal{T} - \Uparrow V) \cap \Uparrow \mathbf{t} \cap \Uparrow \mathbf{f}] \vee temp(b, T) : [\{3\}, \mathcal{T} - \Uparrow \mathbf{t}] \vee T < 60 \leftarrow .$$

4. Resolving the above query with the second fact in DB_3 gives:

$$can\_lift(r1, b) : [\{1,2,3\}, (\mathcal{T} - \Uparrow V) \cap \Uparrow \mathbf{t} \cap \Uparrow \mathbf{f}] \vee temp(b, 61) : [\{3\}, (\mathcal{T} - \Uparrow \mathbf{t}) \cap \Uparrow \mathbf{t}] \vee 61 < 60 \leftarrow .$$

As explained in 2, second and third atoms in the query can be eliminated, leaving us with the query:

$$can\_lift(r1, b) : [\{1,2,3\}, (\mathcal{T} - \Uparrow V) \cap \Uparrow \mathbf{t} \cap \Uparrow \mathbf{f}] \leftarrow .$$

To evaluate this query to the empty clause, we must find the maximal truth value of V that satisfies the following equation$\equiv$ $(\mathcal{T} - \Uparrow V) \cap \Uparrow \mathbf{t} \cap \Uparrow \mathbf{f} = \emptyset$. This is equivalent to$\equiv$ $(\mathcal{T} - \Uparrow V) \cap \{\top\} = \emptyset$ and we conclude that $V = \top$ is the solution to this equation that maximizes the value of V. □

As we can see from the example above, finding the maximum truth value of an annotation variable that enables us to eliminate a query atom results in a maximization problem with some constraints. Each resolution with the atom introduces new restrictions on the set of truth values its annotation variable can legitimately have. Notice that these restrictions can be part of another maximization problem. As an example, suppose we have the following clause in the (regular representation of) DB_1:

$$can\_lift(X, b) : [\{1\}, \Uparrow V_1] \leftarrow can\_lift(X, b) : [\{2\}, \Uparrow V_1].$$

In other words, DB_1 contains the information that DB_2 is a more reliable source of information as far as the object b is concerned. When we resolve this clause with the original query in the above example, we get the following query:

$$can\_lift(r1, b) : [\{1,2,3\}, (\mathcal{T} - \Uparrow V) \cap \Uparrow V_1] \vee can\_lift(X, b) : [\{2\}, \mathcal{T} - \Uparrow V_1] \leftarrow .$$

Here V_1 is going to be maximized as well, and we want to know how the current maximum value of V is affected by the changes in the value of V_1. We are now going to formalize this idea.

5.1 Maximization Problems

Definition 12 (Maximization Problem) let $\mathcal{T}$ be a complete lattice of truth values, $V_1, \ldots, V_n$ be annotation terms and $f_{obj} : \mathcal{T}^n \rightarrow \mathcal{T}$. A maximization problem MP is given as follows:

$$\begin{array}{ll} \textbf{maximize} & f_{obj}(V_1, \ldots, V_n) \\ \textbf{subject to} & T_1 \; \Omega_{1_1} \; f_{1_1}(V_1) \; \Omega_{1_2} \; \ldots \; \Omega_{1_n} \; f_{1_n}(V_n) = \emptyset \\ & \ldots \\ & T_m \; \Omega_{m_1} \; f_{m_1}(V_1) \; \Omega_{m_2} \; \ldots \; \Omega_{m_n} \; f_{m_n}(V_n) = \emptyset \end{array}$$

where $T_i \subseteq \mathcal{T}$, f_{i_j} is a map from $\mathcal{T}$ to $2^{\mathcal{T}}$, and $\Omega_{i_j} \in \{\cap, \cup, \setminus\}$ for all $1 \leq i \leq m, 1 \leq j \leq n$. Intuitively, the expressions on the left of the equalities above are unions/intersections/differences of terms denoting subsets of $\mathcal{T}$.

A mapping $M : \{V_1, \ldots, V_n\} \rightarrow \mathcal{T}$ is said to be an optimal solution to MP iff (1) the assignment of $M(V_i)$ to variable V_i $(1 \leq i \leq n)$ satisfies the constraints and (2) for all other mappings M' that satisfy the constraints, the inequality $f_{obj}(M(V_1), \ldots, M(V_n)) \not\leq f_{obj}(M'(V_1), \ldots, M'(V_n))$ holds w.r.t. the given lattice ordering.

Example 7 Consider the truth value lattice FOUR and suppose we wish to solve the maximization problem

$$\begin{array}{rc} \textbf{maximize} & V_1 \sqcup V_2 \\ \textbf{subject to} & \{\bot, \mathbf{f}\} \cap (\Uparrow V_1) \cap (\Uparrow V_2) = \emptyset \end{array}$$

Then, $V_1 = V_2 = \top$,$V_1 = \top$, $V_2 = \mathbf{t}$ and $V_1 = \mathbf{t}$ $V_2 = \top$ are all solutions to the problem that maximize $V_1 \sqcup V_2$. However, the solution $V_1 = \bot$, $V_2 = \mathbf{t}$ does *not* maximize $V_1 \sqcup V_2$. □

Consider the query $Q^* \equiv \leftarrow A : [D, \mathcal{T} - \Uparrow V_1]$. As has been illustrated in Example 6, when processing this query by performing successive S-resolutions, the atom A (when it occurs in successive resolvents in an S-deduction) will always have an annotation of the form

$$(\mathcal{T} - \Uparrow V_1) \cap (\Uparrow V_2) \cap \ldots \cap (\Uparrow V_n)$$

where $n \geq 1$. When attempting to evaluate the "current best" known truth value for A, we need to maximize the value of V_1 subject to the constraint

$$(\mathcal{T} \setminus \Uparrow V_1) \cap (\Uparrow V_2) \cap \ldots \cap (\Uparrow V_n) = \emptyset.$$

This is because V_1 occurs in the query Q^* and we wish to get the maximal possible value of V_1. Theorem 3 below shows that the optimal solution of this maximization problem is obtained by setting $V_1 = V_2 \sqcup \ldots \sqcup V_n$. Prior to proving Theorem 3, we need to prove an elementary result.

Lemma 1 If $V_1 = V_2 \sqcup \ldots \sqcup V_n$, then $\Uparrow V_1 = (\Uparrow V_2) \cap \ldots \cap (\Uparrow V_n)$.

Proof:

- Since $V_i \leq V_1$ $(2 \leq i \leq n)$, $V_1 \in (\Uparrow V_i)$. Hence for all $V_1 \leq V'$, $V' \in (\Uparrow V_i)$ and $(\Uparrow V_1) \subseteq ((\Uparrow V_2) \cap \ldots \cap (\Uparrow V_n))$.
- Let $V_s = (\Uparrow V_2) \cap \ldots \cap (\Uparrow V_n)$. For all $V' \in V_s$, we have that $V_i \leq V'$ $(2 \leq i \leq n)$. Since $V_1 = V_2 \sqcup \ldots \sqcup V_n$, it must be the case that $V_1 \leq V'$. Hence $V' \in \Uparrow V_1$ and $((\Uparrow V_2) \cap \ldots \cap (\Uparrow V_n)) \subseteq \Uparrow V_1$. □

Theorem 3 For any maximization problem MP given as follows:

$$\begin{array}{rc} \textbf{maximize} & V_1 \\ \textbf{subject to} & (\mathcal{T} \setminus \Uparrow V_1) \cap (\Uparrow V_2) \cap \ldots \cap (\Uparrow V_n) = \emptyset \end{array}$$

where all the V_i,$1 \leq i \leq n$ are annotation terms, the optimal solution is:

$$V_1 = V_2 \sqcup \ldots \sqcup V_n.$$

Proof: The theorem will be proved by induction on the number, n, of annotation variables.

Basis The problem MP_1 be given as follows:

$$\begin{array}{rc} \textbf{maximize} & V_1 \\ \textbf{subject to} & (\mathcal{T} \setminus \Uparrow V_1) = \emptyset \end{array}$$

Then, the optimal solution to MP_1 is $V_1 = \bot$.

- $\sqcup\{\} = \bot$, therefore $V_1 = \bot$ is the solution given in the theorem.
- Since $\Uparrow V_1 = \mathcal{T}$, $(\mathcal{T} \setminus \Uparrow V_1) = \emptyset$ and hence $V_1 = \bot$ is a solution to the constraint given in MP_1.
- There is no solution $V_1^{'}$ such that $\bot \leq V_1^{'}$. Since that implies $\bot \in (\mathcal{T} \setminus \Uparrow V_1^{'})$, $V_1^{'}$ does not satisfy the constraint.

Inductive Step Let for all $i < n$ the solution to the problem MP_i ,

$$\begin{array}{rc} \textbf{maximize} & V_1 \\ \textbf{subject to} & (\mathcal{T} \setminus \Uparrow V_1) \ \cap \ldots \cap \ (\Uparrow V_i) = \emptyset \end{array}$$

be given as $V_1 = V_2 \sqcup \ldots \sqcup V_i$. Let the problem MP_n be :

$$\begin{array}{rc} \textbf{maximize} & V_1 \\ \textbf{subject to} & (\mathcal{T} \setminus \Uparrow V_1) \ \cap \ldots \cap \ (\Uparrow V_n) = \emptyset \end{array}$$

Then the solution to MP_n is $V_1 = V_2 \sqcup \ldots \sqcup V_n$.

- Let $\alpha = V_2 \ \sqcup \ \ldots \ \sqcup V_{i-1}$ and $\beta = \alpha \ \sqcup \ V_i$. By the inductive hypothesis α is a solution to MP_{i-1}. By lemma 1 it is true that

  $$\begin{array}{rcl} \Uparrow \alpha & = & (\Uparrow V_2) \ \cap \ \ldots \ \cap \ (\Uparrow V_{i-1}) \\ (\Uparrow \alpha) \ \cap \ (\Uparrow V_i) & = & (\Uparrow V_2) \ \cap \ \ldots \ \cap \ (\Uparrow V_{i-1}) \ \cap \ (\Uparrow V_i) \end{array}$$

  By lemma 1, $\Uparrow (\alpha \ \sqcup \ V_i) = \ (\Uparrow \alpha) \ \cap \ (\Uparrow V_i) = \Uparrow \beta$. Then,

  $$(\mathcal{T} \setminus \Uparrow \beta) \ \cap \ (\Uparrow V_2) \ \cap \ldots \cap \ (\Uparrow V_i) = \emptyset$$

  and β is a solution to MP_i.

- β is the only solution since for all $V^{'} \not\leq \beta$ is true that $\beta \notin \Uparrow V^{'}$ and $\beta \in (\mathcal{T} \setminus \Uparrow V^{'})$. By the argument above we know that

  $$\begin{array}{rcl} \Uparrow \beta & = & (\Uparrow V_2) \ \cap \ldots \cap \ (\Uparrow V_i) \\ \beta & \in & ((\Uparrow V_2) \ \cap \ldots \cap \ (\Uparrow V_i)) \\ \beta & \in & [(\mathcal{T} \setminus \Uparrow V^{'}) \ \cap \ ((\Uparrow V_2) \ \cap \ldots \cap \ (\Uparrow V_i))] \neq \emptyset \end{array}$$

  Hence, $V^{'}$ doesn't satisfy the constraints for MP_i and cannot be a solution. $\square$

Example 8 Consider the maximization problem:

$$\textbf{maximize } V$$

$$\textbf{subjectto } (\mathcal{T} - V) \cap \Uparrow V_1 \cap \ldots \cap \Uparrow V_{n-1}.$$

The solution to this problem is $V_{old} = V = V_1 \sqcup \ldots \sqcup V_{n-1}$. Now, suppose the term $\Uparrow V_n$ is added to the constraint. Then, the new maximum value of V is $V = V_{old} \sqcup V_n$. In other words, having calculated V_{old} once, we can use it to solve larger problems maximizing the same variable. For instance, in the case of example 6, we had calculated the maximal truth value of V to be $\mathbf{t}$ (in the second step). At step 4, we introduce the term $\Uparrow \mathbf{f}$ into the constraint. Then, the new maximal value of V became $V = \mathbf{t} \sqcup \mathbf{f} = \top$. Therefore we, can conclude that $V = \top$ without solving the maximization problem from scratch. □

When using the above theorem to compute the maximal value of V_1 subject to the constraint that

$$(\mathcal{T} \setminus \Uparrow V_1) \cap (\Uparrow V_2) \cap \ldots \cap (\Uparrow V_n) = \emptyset$$

we need to address how the maximal value of V_1 changes when the value of one of the V_i's changes. The following theorem shows how this may be easily computed.

Theorem 4 Let MP_n be the maximization problem given in Theorem 3 and $V_1 = \alpha = V_2 \sqcup \ldots \sqcup V_n$ be the maximum solution. The problem $MP_n^{'}$ is defined by replacing V_i by $V_i^{'}$ for some $2 \leq i \leq n$ where $V_i \leq V_i^{'}$. The optimal solution to $MP_n^{'}$ is $V_1 = \alpha \sqcup V_i^{'}$.

Proof: Since $\Uparrow V_i \cap \Uparrow V_i^{'} = \Uparrow V_i^{'}$ and by lemma 1, $\Uparrow (\alpha \sqcup V_i^{'}) = \Uparrow \alpha \cap \Uparrow V_i^{'}$, then

$$\begin{array}{rcl} \Uparrow \alpha & = & \Uparrow V_2 \cap \ldots \cap \Uparrow V_n \\ \Uparrow \alpha \cap \Uparrow V_i^{'} & = & \Uparrow V_2 \cap \ldots \cap \Uparrow V_i^{'} \cap \ldots \cap \Uparrow V_n \\ (\mathcal{T} \setminus \Uparrow (\alpha \sqcup V_i^{'})) & \cap & \Uparrow V_2 \cap \ldots \cap \Uparrow V_i^{'} \cap \ldots \cap \Uparrow V_n = \emptyset \end{array}$$

Hence, $V_1 = \alpha \sqcup V_i^{'}$ satisfies the constraint given in $MP_n^{'}$ and it is the maximum such value as a result of the second equality above. □

5.2 Table Organization

The `MULTI_OLDT` table is a linked collection of records. At any given point in time, t, during the processing of query Q, there is a record in the table for each atom that occurs either in Q or in any of the resolvents generated upto that time. Each amalgamated atom in a resolvent generated while constructing one or more deductions points to the corresponding record in the table. The record structure associated with the amalgamated atom $A : [D, V]$ has seven fields described below:

1. **Index:**[6] The index (name) of the annotation variable V. If V is a ground term, then V is replaced by a variable $V^{'}$ both in the query and in the record for $A : [D, V]$. $V^{'}$ must be different from all the variables that appear in the original query and in the intermediate queries as well.

[6] In this paper, we will assume that the annotation functions occur only in clause heads, no variable symbol occurs more than once in an annotation term, and no nesting of annotation functions is allowed.

2. **Known:** The currently known value of the variable under different answer substitutions. This field is a pointer to a linked list of (substitution,truth value) pairs. It is initially set to $(\epsilon, \perp)$. Intuitively, an entry (σ, μ) in the **Known** field of an atom $A : [D, V]$ means that there already exists a refutation for $\forall(A : [D, V])$ with (σ, μ) as a computed (substitution, truth value) pair, i.e. $\forall(A\sigma : [D, \mu])$ is known to be a logical consequence of the program.

3. **Desired:** The minimum truth value necessary to stop further processing of the associated amalgamated atom. This value is $\top$ for all annotation variables (intuitively, this says "the sky ($\top$) is the limit").

4. **Status:** This field is used for expressing control information about the associated atom A. It is set to 0 if V is non-ground. Otherwise, it is set to 1.

5. **Reference:** This field is set as follows: if $A : [D, V]$ was in the original query Q, then this field is `NIL`. Otherwise, it must have been introduced by one and only one clause C involved in a resolution step with an intermediate resolvent Q_i on an atom $A_i : [D_i, V_i]$, in Q_i. In this case, this field points to the entry associated with $A_i : [D_i, V_i]$.

6. **Subsumes:** This field is a pointer to a list of nodes. Each node N in this list contains a pointer to an entry, E, in the table whose associated atom is subsumed by $A : [D, V]$.

7. **Atom:** This field is a pointer to the record storing the atom $A : [D, V]$ in `QUERY` or in `GARBAGE`.

The list `QUERY` is a doubly-linked list of amalgamated atoms to be solved. It has the following properties: (1) Every amalgamated atom is represented by a unique node in the list. (2) Each node stores necessary information about the atoms. (i.e. the predicate symbol, the `DNAME` value, etc.) (3) Each node has a pointer pointing to the table entry associated with the atom stored in it. (4) Nodes have pointers pointing to next and previous entries in the list. In addition to the list `QUERY`, we have the list `GARBAGE` with the same structure. When we want to insert a new atom into `QUERY`, we check if there exists an atom in it that subsumes the new atom. If this is the case, then we don't need to process this atom, and hence, we can insert it into `GARBAGE` instead of into `QUERY`.

5.2.1 Table Creation

Given the regular representation, $Q^* = (A_1 : \texttt{COMP}([D_1, T_1]) \vee \cdots \vee A_m : \texttt{COMP}([D_m, T_m])) \leftarrow$[7] of the (initial) query to an amalgamated `KB`, we create the `MULTI_OLDT` table by calling the the procedure **Create_New_Entry** for all atoms $A_i : \texttt{COMP}([D_i, T_i])$, $1 \leq i \leq m$ as described below:

Insert a node corresponding to the atom $A_i : [D_i, T_i]$ into `QUERY` and call **Create_New_Entry**(T_i). Set a pointer from the node in `QUERY` storing $A_i : [D_i, T_i]$ to the newly created table entry.

<u>function</u> **Create_New_Entry** (T **:annotation term) :address of the table entry**
<u>begin</u>
    allocate a new entry E_{new} for the annotation term
    Set E_{new}.**Reference** and E_{new}.**Subsumes** to `NIL`
    <u>if</u> T is ground <u>then</u>

[7] Without loss of generality, we will assume that the atoms that use the same annotation variables appear consecutively in the query and in all the program clauses. Furthermore, the atoms in the body of a clause appear in the same order as their respective annotation variables appear in the annotation term of the head.

Create a new variable V_{new}, and set E_{new}.**Index** to V_{new}
Set E_{new}.**Desired** to T
else
Set E_{new}.**Status** to 1 and E_{new}.**Known** to $(\epsilon, \perp)$
Set E_{new}.**Index** to T, E_{new}.**Desired** to $\top$
Set E_{new}.**Known** to $(\epsilon, \perp)$
return E_{new}
end

Example 9 Recall example 6. The initial query was:

$$can\_lift(r1, b) : [\{1, 2, 3\}, \mathcal{T} - \Uparrow V] \leftarrow .$$

Initially both the table and the linked list of query atoms are both empty. We insert the node $(can\_lift(r1, b) : [\{1, 2, 3\}, V] -$ Entry 1) into QUERY. The entry in the table corresponding to this atom is as follows:

Index	Known	Desired	Status	Reference	Subsumes	Atom
V	$(\epsilon, \perp)$	$\top$	0	nil	nil	1

$\square$

5.3 Updating The Table Entries During Resolution

When we resolve the (regular representation of) query, Q_j^*, against the clause $C^* =$

$$B_0 : \texttt{EXP}([D_0', f(V_1', \ldots, V_m')]) \quad \leftarrow \quad B_1 : \texttt{EXP}([D_1', V_1'])\& \ldots \& B_n : \texttt{EXP}([D_n', V_n'])$$

on the atom A_i : COMP($[D_i, V_i]$) via mgu θ, two things may happen: (1) some new entries from the body of C^* may need to be added to QUERY and to the table and (2) the (substitution,truth value) pairs associated with the atom A_i : COMP($[D_i, V_i]$) in the table may change. The table needs to be updated to incorporate these changes. These updates are handled as follows:

- For each $1 \leq r \leq n$, we create a new entry, $E(r)$, in the table associated with $B_r\theta$: COMP($[D_r', V_r']$). Two cases may arise, depending upon whether $B_r\theta$: COMP($[D_r', V_r']$) is subsumed[8] by an amalgamated atom associated with an existing entry in the table.
  - **Case 1:** If it is not so subsumed, then the fields of $E(r)$ are set as specified in the Table Creation part above, except that the **Reference** field points to the table entry for atom A_i : COMP($[D_i, V_i]$). The amalgamated atom $B_r\theta : [D_r', V_r']$ is added to QUERY.
  - **Case 2: (Subsumption Check and Cache Usage)** $B_r\theta$: COMP($[D_r', V_r']$) is subsumed by an amalgamated atom, $A^\sharp$ associated with an existing entry $E(A^\sharp)$ in the table. Let γ be a substitution such that $A^\sharp\gamma = B_r\theta$. Then the **Known** field of $E(r)$ is set to (γ,nil). (This pair will be kept to reflect the relationship between already computed pairs in the **Known** field of $E(A^\sharp)$ and $E(r)$.) Furthermore, $B_r\theta : [D_r', V_r']$ is *not added* to the QUERY, instead it is appended to GARBAGE. A pointer to $E(r)$ is added to the list pointed to by

[8] A_1 : COMP($[D_1, V_1]$) subsumes A_2 : COMP($[D_2, V_2]$) iff V_1, V_2 are annotation variables and there is a substitution γ such that $A_1\gamma = A_2$ and $D_1 \subseteq D_2$.

$E(A^\sharp)$.**Subsumes**. For all the pairs (β, μ) in $E(A^\sharp)$.**Known** the following is added to $E(r)$.**Known**: (1) if β' is less general than γ, then add (β', μ), (2) if γ is less general than β', then add (γ, μ). Here β' is obtained from β by throwing away all pairs $X = Y$ such that γ does not contain any substitution for X or Y. The procedure **Copy_Subsumed_Known** which will be given later, is used for copying **Known** field to subsumed entries as explained above. The other fields are set in the same way as specified in the Create_New_Entry algorithm described earlier.

- An additional entry, $E(n+1)$, is added after the entry for $B_n\theta$: `COMP`$([D_n', V_n'])$ If $f(V_1', \ldots, V_m')$ is a ground term, then it is evaluated and its value is stored in the **Desired** field of $E(n+1)$, the **Status** field is set to 1. If $f(V_1', \ldots, V_m')$ is a non-ground annotation term then the address of the code implementing f is stored in the **Index** field and the **Status** is set to 0. The pair (θ,nil) is stored in the **Known** field. The **Reference** and **Subsumes** fields are set to `NIL`.

- The **propagation** of (substitution,truth value) pairs is described below.

Example 10 Consider the first step in Example 6. We resolved the query with the clause:

$$can\_lift : [\{2\}, \Uparrow \mathbf{t}] \leftarrow weight(X, W) : [\{2\}, \Uparrow \mathbf{t}] \,\&\, W < 50.$$

via mgu $\{X = b\}$. At this step, the initial table given in example 9 is modified to:

Index	**Known**	**Desired**	**Status**	**Reference**	**Subsumes**	**Atom**
V	$(\epsilon, \bot)$	$\top$	0	nil	nil	1
V_1	$(\epsilon, \bot)$	**t**	1	Entry 1	nil	2
V_2	$(\epsilon, \bot)$	**t**	1	Entry 1	nil	3
	$(\{X = b\}$,nil)	**t**	1	nil	nil	nil

The atoms in `QUERY` are the following: ($can\_lift(r1, b)$: $[\{1, 2, 3\}, V]$ – Entry 1), ($weight(b, W)$: $[\{2\}, V_1]$ – Entry 2), ($leq(W, 50) : V_2$ – Entry 3). □

5.3.1 Substitution and Truth Value Propagation.

Recall that the entries in the `MULTI_OLDT` table store information concerning the amalgamated atoms in a query. The **Known** field of a given entry stores pairs of the form (θ, μ), which means that if $A : [D, V]$ is pointing to this entry, then $A\theta : [D, \mu]$ is a logical consequence of the program. Assume that the (regular representation of) query A : `COMP` : $[D, V] \leftarrow$ is resolved with the clause A : `EXP`$([D, f(V_1, \ldots, V_m)]) \leftarrow B_1$: `EXP`$([D_1, V_1])$ & ... &B_m : `EXP`$([D_m, V_m])$ and assume that the **Known** fields of the table entries corresponding to the atoms $B_1, \ldots, B_m$ contain the pairs $(\theta_1, \mu_1), \ldots, (\theta_m, \mu_m)$ respectively. From this information we can conclude that $A\theta : [D, f(\mu_1, \ldots, \mu_m)]$ is a logical consequence of the program where θ is any substitution such that it is less general than $\theta_1, \ldots, \theta_m$. In this paper, we will consider a substitution to be a set of equations in solved-form (cf. Martelli and Montanari [25]).

Definition 13 Let θ_1 and θ_2 be two substitutions. σ is said to be the most general common denominator (MGCD) of θ_1 and θ_2 iff

1. σ is less general than both θ_1 and θ_2, i.e. there exists substitutions θ' and θ'' such that $\sigma = \theta_1\theta' = \theta_2\theta''$.

2. For any substitution σ' that satisfies (1), σ' is less general than σ. (It may be the case that σ is less general than σ' as well, in that case σ' belongs to the same equivalence class as σ under the equivalence relation $\sim$ defined as: $\sigma_1 \sim \sigma_2$ iff σ is less general than σ_2 and σ_2 is less general than σ_1.)

Note that if there is a refutation for $(\leftarrow A : [D, \mu])\theta$, then there is a refutation for $(\leftarrow A : [D, \mu])\theta'$ for all θ' less general than θ. The substitutions θ_1 and θ_2 are said to be *compatible* iff there exist substitutions θ' and θ'' such that $\theta_1\theta' = \theta_2\theta''$. Any two compatible substitutions are guaranteed to possess an MGCD and this MGCD is unique upto equivalence.

Proposition 2 Suppose there are refutations for the regular representations, $A_1\theta_1 : \mathtt{COMP}([D_1, \mu_1]) \leftarrow$ and $A_2\theta_2 : \mathtt{COMP}([D_2, \mu_2]) \leftarrow$ where θ_1 and θ_2 are compatible. Then there exists a refutation for the regular representation $A_1\sigma : \mathtt{COMP}([D_1, \mu_1]) \vee A_2\sigma : \mathtt{COMP}([D_2, \mu_2]) \leftarrow$ where σ is an MGCD of θ_1 and θ_2.

Proof. Since there is an S-refutation for both $A_1\theta_1 : \mathtt{COMP}([D_1, \mu_1]) \leftarrow$ and $A_2\theta_2 : \mathtt{COMP}([D_2, \mu_2]) \leftarrow$, it follows, by the Soundness of S-resolution, that both $\forall(A_1\theta_1 : \mathtt{EXP}([D_1, \mu_1]))$ and $\forall(A_2\theta_2 : \mathtt{EXP}([D_2, \mu_2]))$ are logical consequences of the given program P. As σ is less general than both θ_1 and θ_2, it follows that $\forall(A_1\sigma : \mathtt{EXP}([D_1, \mu_1]))$ and $\forall(A_2\sigma : \mathtt{EXP}([D_2, \mu_2]))$ are logical consequences of P as well. By the completeness theorem for S-resolution, there is an S-refutation for the query $\forall(A_1\sigma : \mathtt{EXP}([D_1, \mu_1]) \,\&\, A_2\sigma : \mathtt{EXP}([D_2, \mu_2]))$. $\square$

This result can be extended to queries of arbitrary length. Also note that if we have refutations for $A\theta_1 : \mathtt{COMP}([D, \mu_1]) \leftarrow$ and $A\theta_2 : \mathtt{COMP}([D, \mu_2]) \leftarrow$, and θ_1 and θ_2 are compatible, then there exists a refutation for $A\sigma : \mathtt{COMP}([D, \mu_1 \sqcup \mu_2]) \leftarrow$ where σ is an MGCD of θ_1 and θ_2.

Example 11 Let the truth value lattice be FOUR and assume we have the following program clause C^*:

$$p(X, Y) : \mathtt{EXP}([D, V_1 \sqcap V_2]) \leftarrow q(X, Y) : \mathtt{EXP}([D, V_1]) \ \&\ r(X, Y) : \mathtt{EXP}([D, V_2])$$

Suppose there exist refutations for $q(X, Y)\theta_1 : \mathtt{COMP}([D, \mathbf{t}]) \leftarrow$ and $r(X, Y)\theta_2 : \mathtt{COMP}([D, \mathbf{f}]) \leftarrow$, where $\theta_1 = \{X/a\}$ and $\theta_2 = \{Y/f(Z)\}$. $\sigma = \{X/a, Y/f(Z)\}$ is the MGCD of θ_1 and θ_2, and from this we can conclude that there is a refutation for $p(X, Y)\sigma : \mathtt{COMP}([D, \mathbf{t} \sqcap \mathbf{f}]) \leftarrow$. $\square$

5.3.2 Collecting Truth Values and Substitutions For Refutations

When an S-resolution step is performed and the corresponding maximization problem is solved, the (new) maximal value μ of an annotation variable V is calculated for the mgu θ of the resolution. Hence, (θ, μ) is added to the **Known** list associated with V. V in turn may appear as a constraint in other maximization problems. For example, assume that E is the table entry storing the variable V. If E.**Reference** is not nil, then the atom pointing to E must be in the body of a clause C and C was resolved with the query on atom Q_A. (Q_A is pointing to the table entry pointed to by E.**Reference**.) Then (θ, μ) combined with (substitution, truth value) pairs corresponding to other atoms in the body of C may result in new refutations for Q_A. New refutations must be propagated to Q_A and to all atoms subsumed by Q_A. The table updating process can be summarized in three steps:

1. **Combining Refutations.** (substitution,truth value) pairs corresponding to atoms that occur in the body of the same clause are merged and their their common MGCDs are found.

2. **Updating The Known Field.** Given a list of new (substitution,truth value) pairs, we have to update the **Known** field of the appropriate entry so that the new pairs are incorporated into the **Known** list, and redundant (i.e. subsumed) pairs are eliminated.

3. **Propagating The Updates.** Steps 1 and 2 must be repeated to reflect the effects of the updates on all the atoms. This may be the result of iteration on the **Reference** field or the **Subsumes** field.

Combining Refutations. Recall that entries corresponding to atoms in the body of the same clause C are stored consecutively in the table. We will start from the first entry in the table associated with an atom in C and merge **Known** lists of all the atoms in C. Let us illustrate how this will be done by an example:

Example 12 Let the query atom $Q_A \equiv A : \mathtt{COMP}([D, V])$ be resolved with the clause $A_1 : \mathtt{EXP}([D_1, T_1]) \leftarrow A_2 : \mathtt{EXP}([D_2, T_2]) \,\&\, A_3 : \mathtt{EXP}([D_3, T_3])$ via mgu θ. The atoms in the body of the clause are added to QUERY. Assume, after several resolution steps, that the **Known** lists associated with atoms $A_2 : \mathtt{COMP}([D_2, T_2])$ and $A_3 : \mathtt{COMP}([D_3, T_3])$ contain the pairs (θ_2, μ_2) and (θ_3, μ_3) and σ be the MGCD of θ_2 and θ_3. Then we may conclude the following for Q_A depending on the nature of T_1, T_2 and T_3.

- **Case 1:** T_2, T_3 are variables:
  - **Case 1.1:** T_2 and T_3 are different variables: Then, T_1 must be a function of T_2 and T_3, i.e. $T_1 = f(T_2, T_3)$ for some annotation function f. Then, $A\theta\sigma : COMP([D, f(\mu_2, \mu_3)]) \leftarrow$ has a refutation. The pair $(\theta\sigma, f(\mu_2, \mu_3))$ must be added to the **Known** list of the table entry corresponding to Q_A. The variable S_LIST in procedure Merge_Substitutions below will contain the pair $(\sigma, \langle \mu_2, \mu_3 \rangle)$ at the end of the big while loop and it will contain the pair $(\theta\sigma, f(\mu_2, \mu_3))$ when procedure terminates.
  - **Case 1.2:** T_2 and T_3 are identical. Then, T_1 must also be identical to T_2 and T_3. Then, we must add the pair $(\theta\sigma, \mu_2 \sqcap \mu_3)$ to the **Known** list of Q_A.
- **Case 2:** T_2 is ground and T_3 is a variable: Then, T_1 must be identical to T_3. If $\underline{\text{evaluate}}(T_2) \leq \mu_2$ then we must add the pair $(\theta\sigma, \mu_3)$ to the **Known** list of Q_A. The case when T_3 is ground and T_2 is a variable is similar to this case.
- **Case 3:** T_2, T_3 are both ground: In this case T_1 must also be ground. If $\underline{\text{evaluate}}(T_i) \leq \mu_i$ is true for $i = 2, 3$, then we must add the pair $(\theta\sigma, \underline{\text{evaluate}}(T_1))$ to the **Known** list of Q_A. □

Below, we generalize this idea to clauses of arbitrary length. The variable S_LIST used in the procedure is a list of (substitution, truth value list) pairs. The second list is a list of truth values of all the distinct variables appearing in the clause. When a new variable is encountered during the merging process, its value is appended to the end of the truth value list.

<u>function</u> **Merge_Substitutions (PTR : pointer to table) : list of (substitution,truth value)**
<u>begin</u>
S_LIST $\leftarrow (\epsilon, \langle\,\rangle)$
Previous_Index $\leftarrow$ nil * used to take glb of truth values corr. to same var. in different atoms *\
Ref_Entry $\leftarrow$ PTR.**Reference** * used to find the last entry with the same **Reference** field *\

```
while PTR.Reference = Ref_Entry do
    S_LIST_2 ← NIL
    for all the elements (θ', μ') in PTR.Known do
        for all the elements (θ, ⟨μ_1 ..., μ_i⟩) in S_LIST do
            if there exists MGCD σ of θ and θ' then
              if PTR.Status = 1 and (PTR.Desired ≤ μ') then
                add (σ, ⟨μ_1 ..., μ_i⟩) to S_LIST_2 \* Case 2 in Example 12 *\
              else if PTR.Status = 0 then
                if Previous_Index = PTR.Index then
                  add (σ, ⟨μ_1 ..., μ_{i-1}, μ_i ⊓ μ'⟩) to S_LIST_2 \* Case 1.2 in Example 12 *\
                else
                  add (σ, ⟨μ_1 ..., μ_i, μ'⟩) to S_LIST_2 \* Case 1.1 in Example 12 *\
            end if \* there exists an MGCD of θ_1 and θ_2 *\
    Previous_Index ← PTR.Index
    PTR ← next(PTR)
    S_LIST ← S_LIST_2
end while \* All the entries with the same Reference field *\
\* Now PTR is pointing to the additional entry that stores the substitution of resolution ..... *\
Let PTR.Known be (γ, nil)
for all the elements (θ, ⟨μ_1 ... μ_j⟩) in S_LIST do
    if PTR.Status = 0 then \* call the addressed annotation function *\
        add (γθ, call(#(PTR.Index),⟨μ_1 ... μ_j⟩)) to the output list
    else if PTR.Status = 1 then \* j must be 0 *\
        add (γθ,PTR.Desired) to the output list \* Case 3 in Example 12 *\
    else\* j must be 1 *\
        add (γθ, μ_1) to the output list
end for
Return the output list
end
```

As the reader may have already noticed, this algorithm constantly recomputes the combination of already existing pairs every time new pairs are added to the **Known** field of a table entry. This drawback can easily be remedied by storing the computed (substitution,truth value) pairs in the **Known** field of the additional entry created for all the clauses involved in the resolution. Then, when new pairs are added to the **Known** field of an entry in the table, the merging procedure takes only the substitutions that are compatible with the new substitutions from **Known** fields of the table entries to be merged. (Recall that these entries correspond to atoms that occur in the body of the same clause.) Once all the new pairs are computed, they are merged with the pairs stored as explained above by throwing away the subsumed pairs from each list. This enables us to prune the search space considerably and prevent us from performing the same computations over and over again. Similarly, the propagation procedure will only use the newly computed pairs instead of the whole list of pairs.

Example 13 Consider the second step of Example 6. The table given in example 10 will be updated so that the **Known** fields of second and third entries both contain the pair $(\{W = 19\}, \mathbf{t})$. If we merge these pairs, we get the pair $(\{W = 19\}, \langle\rangle)$ at the end of the big while loop. (recall that the truth values stored in these entries correspond to ground terms, so we don't include them in the truth value list of the pairs in S_LIST.) Since, the **Status** field of the additional entry (Entry 4) is 1, we return the pair $(\{W = 19\}, \mathbf{t})$ at the end of the merging procedure. □

Propagating The Updates. This phase has several different functions. Recall that we had resolved a clause C with the query on an atom (this atom points to the table entry PTR). First, we may need to add new atoms to `QUERY` (or to `GARBAGE`) if the body of C is not empty. The procedure Store_Atoms is used for this purpose and it calls a function named Check_Loop for all the atoms to check if any of the atoms causes a positive loop in the query. If this is the case, all the changes corresponding to the last resolution step are deleted and the table is restored. If C is a fact, then we have obtained a new (substitution,truth value) pair for the entry PTR. Then, the procedure Update_Table updates the **Known** field of PTR by inserting this pair and reflects the changes in PTR.**Known** to the table entry PTR.**Reference** by merging the **Known** fields as explained in example 12 and calling the procedure Update_Known repeatedly until no new pairs can be found. This process is expanded to the entries that are subsumed by an entry that has been updated. If PTR.**Known** has been updated, then new **Known** fields are calculated for all the entries subsumed by PTR by calling the procedure Copy_Subsumed_Known.

```
procedure Update_Table (PTR : pointer to a table entry
                        C : Clause the query atom is resolved with
                        θ : unifying substitution)

\* PTR is the table entry corresponding to the query atom that has been resolved *\
begin
    if body of C is not empty then
      Store_Atoms (PTR,C,θ)
      \* Store the atoms in the body C to QUERY and create corr. table entries *\
    else
      Let C be A : EXP([D,T]) ← \* T must be ground *\
      Update_Known ( (θ, evaluate(T),PTR) \* New refutation found, insert into Known *\
      Update_LIST ← PTR + PTR.Subsumes
      \* Update list is a list of pointers pointing to table entries that have been updated *\
      \* These updates must be propagated *\
      while Update_LIST <> nil do
            PTR ← Update_LIST.PTR \* set PTR to the first pointer in the list *\
            Update_LIST ← next(Update_LIST) \* Discard the first element in the list *\
            while PTR.Reference <> nil do
                  while PTR.Reference = previous(PTR). Reference do
                        PTR ← previous(PTR)
                        \* find the first table entry with same Reference field as PTR *\
                  S_LIST ← Merge_Substitutions (PTR)
                  if S_LIST = nil then
                    Update_LIST ← next(Update_LIST)
                    PTR ← Update_LIST.PTR
                  else
                    PTR ← PTR.Reference
                    Update_Known (S_LIST,PTR)
                    if no new pairs are inserted to PTR.Known then
```

```
                        Update_LIST ← next(Update_LIST)
                      else
                        Update_LIST ← Update_LIST + PTR.Subsumes
                        Subsumes_LIST ← PTR.Subsumes
                        while Subsumes_LIST <> nil then
                              Subsumes_PTR ← Subsumes_LIST.PTR
                              Copy_Subsumed_Known (PTR.Known, S_PTR.Known)
                              Subsumes_LIST ← next(Subsumes_LIST)
                        endwhile \* Copy the Known list to subsumed entries *\
              endwhile \* PTR.Reference is nil, move to next element in Update_LIST *\
      endwhile \* Update_LIST is empty, all updates are propagated *\
end
```

procedure Copy_Subsumed_Known (K1,K2:Known lists)

```
\* Atom with Known list K1 subsumes the atom with Known list K2 *\
begin
      Let (γ, ⊥) be the first pair in K2
      for all pairs (θ, μ) in K1 do
          throw away all pairs X = Y from θ
          such that there is no pair X = X' or Y = Y' in γ for some X' or Y'
          and obtain θ'
          if θ' is less general than γ then
             Create a primary node PTR for Known list⁹ for (θ, μ)
             INSERT (K2,PTR) \* The procedure is given later in the paper *\
          else if γ is less general than θ' then
             Create a primary node PTR for (ε, μ)
             INSERT (K2,PTR)
      Insert a primary pointer for (γ, ⊥) to the beginning of K2
end
```

The following example illustrates the working of the **Copy_Subsumed_Known** routine.

Example 14 Assume the atom $Q_{A_1} \equiv p(X,Y) : \texttt{COMP}([\{1,2\},V_1])$ is part of a query Q' and after several steps of resolution we encounter the subquery $Q_{A_2} \equiv p(X,f(X)) : \texttt{COMP}([\{1,2,3\},V_2])$. Q_{A_1} subsumes Q_{A_2} since $p(X,Y)\gamma = p(X,f(X))$ for $\gamma = \{Y = f(X)\}$ and $\{1,2\} \subseteq \{1,2,3\}$.

Suppose the following pairs are stored in the **Known** list of the entry associated with Q_{A_1}:

$$((\{X=Y\},\mathbf{t}),(\{X=b\},\mathbf{f}),(\{Y=b\},\mathbf{t}),(\{X=f(Y)\},\mathbf{f}),(\{Y=f(a)\},\top).)$$

Then we copy the following pairs to the **Known** list of the entry associated with Q_{A_2}:

$$(-,(\{X=b\},\mathbf{f}),-,-,-,(\{X=a\},\top).)$$

The $-$ above denotes a "don't care" symbol. □

procedure Store_Atoms (PTR : pointer to a table entry
C : Clause the query atom is resolved with
θ : unifying substitution)

```
\* PTR is the table entry corresponding to the query atom that has been resolved *\
begin
    Subsuming_Entries ← nil \* list of table entries that subsume a new atom *\
    LOOP_FLAG ← false \* set if a positive loop is detected *\
    Last_Table_Entry ← pointer to the last entry in the table
    Last_Query_Atom ← pointer to the last node in QUERY
    Last_Garbage_Atom ← pointer to the last node in GARBAGE
    for all atoms A in the body of C do
      Create an entry NEW_PTR in the table
      NEW_PTR.Reference ← PTR
      for all atoms B in QUERY that subsume A do
        Let PTR1 point to the table entry corresponding to B
        Subsuming_Entries ← Subsuming_Entries + PTR1
        add a pointer to PTR1.Subsumes pointing to NEW_PTR
      LOOP_FLAG ← Check_loop (NEW_PTR,Subsuming_Entries)
      if LOOP_FLAG then
        Remove all the entries corresponding to C from the table using Last_Table_Entry
        Delete the Subsumes entries pointing to an atom in C using Subsuming_Entries
        Remove all atoms added to QUERY and GARBAGE from C
          using Last_Query_Atom and Last_Garbage_Atom
        HALT
      else
        add (A – NEW_PTR) to GARBAGE
    else
      add (A – NEW_PTR) to QUERY
    endfor
    create the last entry as explained in Section 5.3
end
```

The following procedure checks for positive loops. It uses the list of entries that subsume the new entry which was constructed in procedure **Store_Atoms**. We are inserting the atom A associated with the table entry PTR. We detect a positive loop if any of the entries reachable from PTR following the **Reference** pointers corresponds to an atom that subsumes A or is subsumed by A.

procedure Check_loop (PTR:Subsumed Entry,
Subsuming_Entries: List of Subsuming Entries)

```
\* PTR is a pointer to a table entry associated with an atom.
Subsuming_Entries is a list of pointers to table entries
associated with atoms that subsume the atom associated with PTR
the function head returns the first pointer to a table entry in the list *\
Q_A ← PTR.Atom
PTR ← PTR.Reference
while PTR <> nil do
```

```
        TEMP_Subsuming ← Subsuming_Entries
        while TEMP_Subsuming <> nil   do
            if PTR = head(TEMP_Subsuming)   then
                HALT, return true
            TEMP_Subsuming ← next(TEMP_Subsuming)
        endwhile
        if PTR.Atom subsumes Q_A then
            HALT, return true
        PTR ← PTR.Reference
    endwhile
    return false \* if the procedure reaches this point without terminating
    then, it means that no positive loop is detected  *\
end
```

The following example shows how the loop checking procedure works.

Example 15 Suppose program P consist of the single clause C^*:

$$p(X) : \texttt{EXP}([\{s\}, \mathbf{t}]) \leftarrow p(X) : \texttt{EXP}([\{s\}, \mathbf{t}])$$

and we ask the query Q^*:

$$p(X) : \texttt{COMP}([\{\mathbf{s}\}, \mathbf{t}]) \leftarrow$$

We create a table entry E_1 for the query atom, and set E_1.**Known** to ($\perp$,nil). We then resolve Q^* with C^*. We create an entry E_2 in the table for the new atom, and set E_2.**Reference** to E_1. Before we insert the new atom $A_{new} = p(X) : \texttt{COMP}([\{\mathbf{s}\}, \mathbf{t}])$, we check if it is subsumed by an atom in the query. Since the single atom in Q^* subsumes A_{new}, we decide not to insert it. Instead, we create a pointer pointing to E_2 and insert it into the E_1.**Subsumes** list. The list Subsuming_Entries is set to E_1. Then, we check for positive loops by calling **Check_loop**. **Check_loop** traverses all the entries reachable from E_2 by following the the **Reference** pointers and checks if any of them is in the Subsuming_Entries list. Since E_2.**Reference** is equal to E_1 and E_1 is in the Subsuming_Entries list, we detect a positive loop. Hence, we delete E_2 and all the entries pointing to E_2(i.e. the pointer in E_1.**Subsumes** pointing to E_2).

Suppose the following fact C_2^* had also been in the program:

$$p(a) : \texttt{EXP}([\{\mathbf{s}\}, \mathbf{t}]) \leftarrow$$

Then, we resolve Q^* with C_2^* to obtain the (substitution,truth value) pair $(\{X = a\}, \mathbf{t})$ and insert it into E_1.**Known**. □

Updating The Known Field. Upto now, we treated the **Known** field as if it were a simple linked list. As explained in the preceding sections, after a resolution step, we may find new (substitution,truth value) pairs (θ, μ) for an atom $A : [D, V]$ which means that $A\theta : [D, \mu]$ is a logical consequence of the program. We then need to update the **Known** field of the table entry corresponding to this atom. The procedure **Update_Known** is invoked by the list of new (substitution,truth value) pairs to be inserted and a pointer to the table entry we're updating. For all pairs in the list, it checks to see if they are subsumed by an existing pair in the **Known** structure, or if they subsume any existing entries. In other words, the procedure checks if any of the cases described below occurs and performs the appropriate actions. Before we give the description of the procedure **Update_Known**, we will develop data structures to maintain the **Known** field efficiently together with explanations of the cases that may arise during the execution of the procedure.

5.3.3 Managing (Substitution-Truth Value) Pairs and Updating The Known Field

We introduce a data structure to maintain the **Known** field of the table. The main objective is to be able to update the **Known** field efficiently, given a doubly linked list of nodes that contains information regarding the new (substitution,truth value) pairs to be inserted into the structure. We want to avoid making unnecessary calls to the MGCD function as well as to reduce the number of variant checks. The data structure therefore will be a linked list where each node (we will refer to these nodes as being primary nodes) stores information about (substitution,truth value) pair, the node's location and it has a field that points to a secondary linked list with a different structure. (we will refer to the nodes in the secondary list as secondary nodes.) This secondary list enables a primary node storing a substitution value θ to access all primary nodes storing substitution values less general than θ.

Primary Node Structure. A primary node is a record containing the following information:

1. **Pointers:**
   (a) **Next:** The next entry in the primary list.
   (b) **Prev:** The previous entry in the primary list.
   (c) **LG:** This field is a pointer to a secondary linked list whose structure is given below and its entries point to primary nodes that store substitution values less general than the **Subs** field (defined below).
2. **Info:** The information is stored in the following fields:
   (a) **Subs:** The substitution stored in the node.
   (b) **TV:** The truth value associated with this substitution.
   (c) **Top:** Number of secondary nodes in the structure that point directly to the primary node.
3. **Status Bits:**
   (a) **Updated:** Set if the **TV** field is changed during a pass of the procedure UPDATE_ALL.
   (b) **Deleted:** Set if the node is deleted (a node is not deleted physically if **Top** is not 0).
   (c) **Scanned:** Toggled if the node is accessed during updates.

Secondary Node Structure. Secondary lists are pointed to by the **LG** field of nodes in the primary list and it has the following structure:

1. **Next_LG:** Pointer to the next entry in the secondary list.
2. **Prim_LG:** Pointer to a primary node storing a **Subs** value less general than the **Subs** value of the current primary node.

Assume N_1, N_2 are primary nodes in the data structure containing (substitution,truth value) pairs (θ_1, μ_1) and (θ_2, μ_2) respectively and θ_2 is less general than θ_1. Then, the following is true initially (i.e. when the data structure does not contain any nodes) and it will be maintained as an invariant by all operations that will be defined on the data structure.

1. N_2 is reachable from N_1. (A primary node A is reachable from another primary node B iff the **LG** list of B contains a secondary node with the **Prim_LG** field pointing to A, or a secondary node in the **LG** list of B points to C and A is reachable from C.)

2. $\mu_1 \leq \mu_2$.

3. N_2 occurs to the right of N_1 in the primary linked list.

5.3.4 Operations on the Data Structure

Assume we are inserting the (substitution,truth value) pair (θ, μ) into the data structure and the structure already contains a node storing the pair (θ', μ'). Depending on the values of (θ, μ) and (θ', μ'), we may need to make changes in the primary and secondary lists. The operations we will use to make these changes are as follows:

- **Case 1:** If θ' is less general than θ and $\mu' \leq \mu$, then the new (substitution,truth value) pair subsumes the existing node. In that case, we must delete the existing node. Note, that we cannot delete the node physically if there are secondary nodes pointing to it (corresponding to the case where **Top** is non-zero. In this case, we just mark it deleted).

  procedure DELETE(PTR:primary node)

  ```
  Set PTR.Deleted to 1
  if PTR.Top ≠ 0 then HALT
  else
    LGPTR ← PTR.LG
    while LGPTR ≠ nil do
          P1 ← LGPTR.Prim_LG
          Decrease P1.Top by one
          if P1 was marked Deleted then
            DELETE(P1)
          LGPTR ← LGPTR.Next_LG
    FREE (PTR.LG) \* return the secondary list associated with PTR to available storage *\
    Set (PTR.Prev).Next to PTR.Next
    Set (PTR.Next).Prev to PTR.Prev
    FREE (PTR)
  end
  ```

- **Case 2:** If θ' is less general than θ and $\mu \not\leq \mu'$, then we have to update the truth value associated with θ' to $\mu \sqcup \mu'$. We know that any (substitution, truth value) pair (θ'', μ'') stored in the structure such that θ'' is less general than θ' will be less general than θ, then we have to update all such pairs as described above and we should delete any pairs that are subsumed by a primary node.

```
procedure UPDATE_ALL(PTR:primary node,μ:truth value)

LGPTR ← PTR.LG
while LGPTR ≠ nil do
      P1 ← LGPTR.Prim_LG
      if P1 was marked Deleted or P1.TV ≤ μ then
        Remove the secondary node pointed to by LGPTR from PTR.LG
        LGPTR2 ← P1.LG
        while LGPTR2 ≠ nil do
              ADD_SECONDARY_PTR (PTR,LGPTR2.Prim_LG)
              LGPTR2 ← LGPTR2.Next_LG
        Decrease P1.Top by 1
        if P1.Top = 0 then
          DELETE(P1)
      else
        Set P1.TV to P1.TV ⊔ μ
        Set P1.Updated to 1
        UPDATE_ALL (P1,P1.TV)
      LGPTR ← LGPTR.Next_LG
end
```

- **Case 3:** If θ' is less general than θ and $\mu \leq \mu'$, then using the same argument in **Case 2**, all the primary nodes storing substitution values less general than θ' have to be updated. Since we know that these nodes will have a corresponding truth value greater than μ by invariant 2 given in section 5.3.3, the update will have no effect on their **TV** field. For this reason, we just mark them all **Updated**.

  Note that during a pass of the algorithm, we may find a primary node with **Subs** value more general than θ and **TV** field incomparable with μ. Then, we have to change μ by taking *lub* with this value. This means that, some of the nodes that were not effected by μ may be effected by its new value. To reflect these changes, once the primary list is totally scanned, the **TV** field of all the nodes that were updated will be changed by taking *lub* with the last value of μ.

```
procedure MARK_ALL(PTR:primary node)

Set PTR.Updated to 1
LGPTR ← PTR.LG
while LGPTR ≠ nil do
      P1 ← LGPTR.Prim_LG
      if P1 was marked Deleted then
        Remove the secondary node pointed to by LGPTR from PTR.LG
        LGPTR2 ← P1.LG
        while LGPTR2 ≠ nil do
              ADD_SECONDARY_PTR (PTR,LGPTR2.Prim_LG)
              LGPTR2 ← LGPTR2.Next_LG
        Decrease P1.Top by 1
```

```
      if P1.Top = 0 then
         DELETE(P1)
    MARK_ALL(LGPTR.Prim_LG)
    LGPTR ← LGPTR.Next_LG
end
```

- **Case 4:** After we update the table using the operations described above, we may still need to insert the pair (θ, μ) since it is not subsumed by any pair in the structure. We may also need to insert a new pair $(\sigma, \mu \sqcup \mu')$ where σ is the MCGD of θ and θ', in the case when neither θ nor θ' are less general than each other and $\mu \not\leq \mu'$. Then we can only conclude $(\sigma, \mu \sqcup \mu')$ from these two pairs.

  To insert a primary node PTR, we have to make sure that PTR is reachable from all the primary nodes with **Subs** field more general than PTR.**Subs**. Similarly, all the nodes storing **Subs** values less general than PTR.**Subs** must be reachable from PTR. To achieve that, we check all the primary nodes in the list and adjust the secondary lists as described below. We make use of the variable LAST_PTR which shows the rightmost primary pointer PTR1 such that PTR1 is more general than PTR.**Subs**.

  procedure INSERT (Known,PTR:primary node)

```
Set LAST_PTR,PTR1 to Known
while PTR1 ≠ nil do
    if PTR1 is not Scanned then
      if PTR.Subs is less general than PTR1.Subs then
        if PTR.TV ≤ PTR1.TV th en
          if PTR.Top = 0 then
            HALT \* PTR1 subsumes PTR, do not insert PTR *\
          else \* PTR.Top ≠ 0 *\
            Set PTR.Deleted to 1
            Insert PTR to the primary list after LAST_PTR
            HALT
        else \* PTR.TV ≰ PTR1.TV *\
          ADD_SECONDARY_PTR (PTR1,PTR)
          Set LAST_PTR to PTR1
      else if PTR1.Subs is less general than PTR.Su bs then
        if PTR1.TV ≤ PTR.TV then
          DELETE(PTR1)
        else
          ADD_SECONDARY_PTR (PTR,PTR1)
          MARK_ALL(PTR1)
      end if \* PTR1 is not Scanned *\
    PTR1 ← PTR1.Next
    end while
Insert PTR in the primary linked list right after LAST_PTR
if LAST_PTR = Known then
```

Set **Known** to PTR
<u>end</u>

- The following procedure is called when PTR2.**TV** is less general than PTR1.**TV**, where PTR1 and PTR2 are primary nodes. It creates a secondary pointer that points to PTR2 and inserts it into PTR1.**LG**. Since this operation results in an additional secondary node pointing to PTR2, PTR2.**Top** field is increased by 1.

  <u>procedure</u> ADD_SECONDARY_PTR (PTR1,PTR2: Primary nodes)

  * PTR2.**TV** is less general than PTR2.**TV** *\
  Create secondary pointer LGTEMP
  Set LGTEMP.**Prim_LG** to PTR2
  Insert LGTEMP into PTR1.**LG**
  Increase PTR2.**Top** by 1 <u>end</u>

Example 16 Let $\mathcal{T}$=FOUR and the query Q^* be given as follows: $p(X,Y) : [\{\mathbf{s}\}, \mathcal{T} \setminus V] \leftarrow$. Let the **Known** field of the table entry corresponding to this atom contain the following (substitution,truth value) pair:

1. **Subs**=$\{X = a\}$, **TV**=**t**, **Top**=0, **LG**= nil

Next we insert the pair $(\{X = Z, Y = f(Z)\}, \mathbf{f})$. The list becomes:

1. **Subs**=$\{X = a\}$, **TV**=**t**, **Top**=0, **LG**= 2, 3
2. **Subs**=$\{X = Z, Y = f(Z)\}$, **TV**=**f**, **Top**=1, **LG**=nil
3. **Subs**=$\{X = a, Y = f(a)\}$, **TV**=$\top$,**Top**=1, **LG**= nil

Next, we insert $(\{X = Z, Y = f(a)\}, \mathbf{t})$. The **Known** field changes to:

1. **Subs**=$\{X = a\}$, **TV**=**t**, **Top**=0, **LG**= 2, 4
2. **Subs**=$\{X = Z, Y = f(Z)\}$, **TV**=**f**, **Top**=1, **LG**= 3
3. **Subs**=$\{X = Z, Y = f(a)\}$, **TV**=**t**, **Top**=1, **LG**=nil
4. **Subs**=$\{X = a, Y = f(a)\}$, **TV**=$\top$,**Top**=1, **LG**=n il, **Deleted**

Finally, we insert $(\{X = a, Y = f(Z)\}, \top)$. The final structure is:

1. **Subs**=$\{X = a\}$, **TV**=**t**, **Top**=0, **LG**= 2
2. **Subs**=$\{X = Z, Y = f(Z)\}$, **TV**=**f**, **Top**=1, **LG**= 3, 4
3. **Subs**=$\{X = Z, Y = f(a)\}$, **TV**=**t**, **Top**=1, **LG**=nil
4. **Subs**=$\{X = a, Y = f(Z)\}$, **TV**=$\top$, **Top**=1, **LG**= nil

$\square$

The following procedure updates the **Known** list of an atom after an S-resolution step has been performed.

procedure Update_Known (S_LIST : a list of (substitution,truth value) pairs,
PTR: pointer to the annotation table)

```
for all (θ, μ) in S_LIST do
    restrict θ to the variables appearing in PTR.Atom.
    Set CHANGED to false \* CHANGED is a boolean variable, it shows if μ is changed *\
    LAST_PTR,KPTR ← PTR.Known
        \* LAST_PTR is as described above,
         KPTR is a temporary pointer used for iterating on PTR.Known *\
    Create a primary node KNEW for (θ, μ)
    for all (θ', μ') in KPTR that is not marked Scanned do
        Initialize KPTR.Updated to 0
        Toggle KPTR.Scanned
        if ∃ MGCD σ of θ and θ' then
          if σ is a variant of θ' then   \* θ' is less general than θ *\
            if μ ≤ μ' then \* Case 2 of section 5.3.4 *\
              MARK_ALL(KPTR)
              ADD_SECONDARY_PTR (KNEW,KPTR)
            else if μ' ≤ μ then
              DELETE (KPTR) \* Case 1,2 *\
              UPDATE_ALL (KPTR,μ)
            else  \* μ, μ' are incomparable *\
              Set KPTR.TV to KPTR.TV ⊔ μ
              Set KPTR.Updated to 1
              UPDATE_ALL (KPTR,KPTR.TV) \* Case 2 *\
              ADD_SECONDARY_PTR (KNEW,KPTR)
          else if σ is a variant of θ then   \* θ is less general than θ' *\
            if μ ≤ μ' then
              Insert KNEW after LAST_PTR
              DELETE(KNEW) \* the new pair is subsumed by some entry in the structure *\
              EXIT inner for loop
            else  \* μ ≰ μ' *\
              ADD_SECONDARY_PTR (KPTR,KNEW)
              LAST_PTR ← KPTR
              Set KNEW.TV to KNEW.TV ⊔ μ'
              Set CHANGED to true, μ to KNEW.TV
          else  \* θ, θ' are incomparable *\
            Create a primary node KNEW_MGCD for (σ, μ ⊔ μ')
              \* Recall that σ is MGCD of θ and θ' *\
            Set KNEW_MGCD.Updated to 1
```

Check if σ is already in the primary list by traversing down the **LG** list of KPTR
If so, set its **TV** field by taking lub with $\mu \sqcup \mu'$
If σ is not in the list
Put KNEW_MGCD right after KNEW * it will be inserted later *\
<u>end of inner loop</u>
<u>if</u> CHANGED <u>then</u> * μ has changed, modify all the entries marked **Updated** by taking lub *\
KPTR $\leftarrow$ PTR.**Known**
<u>while</u> KPTR $\neq$ nil <u>do</u>
<u>if</u> KPTR is not marked **Scanned** <u>then</u>
Set KPTR.**TV** to KPTR.**TV** $\sqcup \mu$
KPTR $\leftarrow$ KPTR.**Next**
<u>for</u> all KNEW_MGCD coming after KNEW <u>do</u> * insert all the new pairs generated by MGCDs *\
INSERT (**Known**,KNEW_MGCD)
Insert KNEW just after LAST_PTR
<u>end</u>

The following example shows how the **Known** list is updated once an S-resolution step is performed.

Example 17 Consider example 6. We specify below, the information stored in the table, the `QUERY` and the `GARBAGE` lists immediately before the last step of example 6 is completed. `QUERY` $\equiv$ $(can\_lift(r1,b) : [\{1,2,3\},V]$ – Entry 1), $(weight(b,W) : [\{2\},V_1]$ – Entry 2), $(leq(W,50) : V_2$ – Entry 3), $(temp(b,T) : [\{3\},V_3]$ – Entry 5), $(le(T,60) : V_4$ – Entry 6), `GARBAGE` $\equiv ()$, the table:

Index	Known	Desired	Status	Reference	Subsumes	Atom
V	$(\epsilon, \mathbf{t})$	$\top$	0	nil	nil	1
V_1	$(\epsilon,\perp),(\{W=19\},\mathbf{t})$	**t**	1	Entry 1	nil	2
V_2	$(\epsilon,\perp),(\{W=19\},\mathbf{t})$	**t**	1	Entry 1	nil	3
	$(\{X=b\}$,nil)	**t**	1	nil	nil	nil
V_3	$(\epsilon,\perp)$	**f**	1	Entry 1	nil	4
V_4	$(\epsilon,\perp)$	**t**	1	Entry 1	nil	5
	$(\{X'=b\}$,nil)	**t**	1	nil	nil	nil

When step 4 is completed, `QUERY` and `GARBAGE` remain the same, but the table is updated to obtain:

Index	Known	Desired	Status	Reference	Subsumes	Atom
V	$(\epsilon, \top)$	$\top$	0	nil	nil	1
V_1	$(\epsilon,\perp),(\{W=19\},\mathbf{t})$	**t**	1	Entry 1	nil	2
V_2	$(\epsilon,\perp),(\{W=19\},\mathbf{t})$	**t**	1	Entry 1	nil	3
	$(\{X=b\}$,nil)	**t**	1	nil	nil	nil
V_3	$(\epsilon,\perp),(\{T=61\},\mathbf{f})$	**f**	1	Entry 1	nil	4
V_4	$(\epsilon,\perp),(\{T=61\},\mathbf{t})$	**t**	1	Entry 1	nil	5
	$(\{X'=b\}$,nil)	**t**	1	nil	nil	nil

$\square$

6 Related Work

A great deal of work has been done in *multidatabase* systems and *interoperable* database systems[39, 16, 36]. However, most of this work combines standard relational databases (no deductive capabilities). Not much has been done on the development of a semantic foundation for such databases. The work of Grant et. al. [16] is an exception: the authors develop a calculus and an algebra for integrating information from multiple databases. This calculus extends the standard relational calculus. Further work specialized to handle inter-operability of multidatabases is critically needed. However, our paper addresses a different topic – that of integrating multiple deductive databases containing (possibly) inconsistencies, uncertainty, non-monotonic negation, and possibly even temporal information. Zicari et. al [39] describe how interoperability may be achieved between a rule-based system (deductive DB) and an object-oriented database using special *import/export* primitives. No formal theory is developed in [39]. Perhaps closer to our goal is that of Whang et. al. [36] who argue that Prolog is a suitable framework for schema integration. In fact, the approach of Whang et. al. is in the same spirit as that of metalogic programming discussed earlier. Whang et. al. do not give a formal semantics for multi-databases containing inconsistency and/or uncertainty and/or non-monotonicity and/or temporal information.

Baral et. al. [2, 3] have developed algorithms for combining different logic databases which generalizes the update strategy by giving priorities to some updates (when appropriate) and as well as not giving priorities to updates (which corresponds to combining two theories without any preferences). Combining two theories corresponds, roughly, to finding maximally consistent subsets (also called flocks by Fagin et. al. [13, 14]). As we have shown in [32], our framework can express maximal consistency as well. [2, 3] do not develop a formal model-theoretic treatment of combining multiple knowledge bases, whereas our method does provide such a model theory. [2, 3] are unable to handle non-monotonicity (in terms of stable/well-founded semantics), nor uncertainty, nor time-stamped information – our framework is able to do so.

Dubois, Lang and Prade [12], also suggest that formulas in knowledge bases can be annotated with, for each source, a lower bound of a degree of certainty associated with that source. The spirit behind their approach is similar to ours, though interest is restricted to the $[0, 1]$ lattice, the stable and well-founded semantics are not addressed, and amalgamation theorems are not studied. However, for the $[0, 1]$ case, their framework is a bit richer than ours when nonmonotonic negations are absent.

In [15], Fitting generalizes results in [35, 4], to obtain a well-founded semantics for bilattice-based logic programs. We have given a detailed comparison of our declarative framework with Fitting's in [32].

Warren and his co-workers [10, 9] have studied OLDT-resolution for ordinary logic programs (both with,and without nonmonotonic forms of negation). In this paper, we have dealt only with the monotonic case, and have focused on (1) how truth value estimates of atoms can be monotonically improved as computation proceeds and how this monotonic improvement corresponds to solving certain kinds of incremental optimization problems over a lattice domain, (2) how OLDT tables must be organized so as to efficiently support such computations. As OLDT-resolution is known to be closely related to magic set computations, we will not discuss those separately.

7 Conclusions

Wiederhold has proposed mediators as a framework within which multiple databases may be integrated. In the first of this series of papers [32], it has been shown that certain forms of annotated logic provide a simple language within which mediators can be expressed. In particular, it was shown that the semantics of "local" databases can be viewed as embeddings within the semantics of amalgamated databases.

In [32], we did not develop an operational theory for query processing in amalgamated KBs. In this paper, we have provided a framework for implementing such a query processing paradigm. This framework supports:

- *incremental, approximate query processing* in the sense that truth value estimates for certain atomic queries will increase as we continue processing the query. Thus if a user (or a machine) wishes to interrupt the processing, then at least an approximate estimate will be obtained,based on which a knowledge based system may take some actions.
- *reuse of previous computations* using the table data structure(s). In particular, we have specified access paradigms for updating answers, i.e. (substitution, truth-value) pairs as processing continues.

In future work, we will extend the above paradigm to handle non-monotonic modes of negation. We are also in the process of starting an implementation of the above paradigm.

Acknowledgements. We have benefited from conversations with Mike Kifer, Jim Lu and Terry Swift.

References

[1] R. Agrawal, R. Cochrane and B. Lindsay. (1991) *On Maintaining Priorities in a Production Rule System*, Proc. VLDB-91, pps 479–487.

[2] C. Baral, S. Kraus and J. Minker. (1991) *Combining Multiple Knowledge Bases*, IEEE Trans. on Knowledge and Data Engineering, 3, 2, pps 200-220.

[3] C. Baral, S. Kraus, J. Minker and V.S. Subrahmanian. (1992) *Combining Knowledge Bases Consisting of First Order Theories*, Computational Intelligence, 8, 1, pps 45–71.

[4] C. Baral and V.S. Subrahmanian. (19910 *Dualities between Alternative Semantics for Logic Programming and Nonmonotonic Reasoning*, Proc. 1991 Intl. Workshop on Logic Programming and Nonmonotonic Reasoning, MIT Press. Full version in: Journal of Automated Reasoning, 10, pps 339–420, 1993.

[5] F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman. (1986) *Magic Sets and Other Strange Ways to Implement Logic Programs*, Proc. 5th Symp. on Principles of Database Systems, pps 1–15.

[6] C. Beeri and R. Ramakrishnan. (1987) *On the Power of Magic*, Proc. 6th Symp. on Principles of Database Systems, pps 269–283.

[7] H. A. Blair and V.S. Subrahmanian. (1987) *Paraconsistent Logic Programming*, Theoretical Computer Science, 68, pp 35-54. Preliminary version in: LNCS 287, Dec. 1987, Springer.

[8] Y. Breitbart, H. Garcia-Molina and A. Silberschatz. (1992) *Overview of Multidatabase Transaction Management*, VLDB Journal, 2, pps 181–239.

[9] W. Chen and D.S. Warren. (1992) *A Goal-Oriented Approach to Computing Well-Founded Semantics*, Proc. 1992 Intl. Conf. on Logic Programming (ed. K.R. Apt), MIT Press.

[10] S. Dietrich and D.S. Warren. (1986) *Extension Tables: Memo Relations in Logic Programming*, SUNY Stonybrook Tech. Report 86/18.

[11] D. Dubois, J. Lang and H. Prade. (1991) *Towards Possibilistic Logic Programming*, Proc. 1991 Intl. Conf. on Logic Programming, ed. K. Furukawa, pps 581–595, MIT Press.

[12] D. Dubois, J. Lang and H. Prade. (1992) *Dealing with Multi-Source Information in Possibilistic Logic*, Proc. 10th European Conf. on Artificial Intelligence, Wiley.

[13] R. Fagin, J.D. Ullman, and M.Y. Vardi. (1983) *On the Semantics of Updates in Databases*, Proc. ACM SIGACT/SIGMOD Symposium on Principles of Database Systems, pps 352–365.

[14] R. Fagin, G. Kuper, J. Ullman, and M. Vardi. (1986) *Updating Logical Databases*, In *Advances in Computing Research*, volume 3, pages 1–18, 1986.

[15] M. C. Fitting. (1991) *Well-Founded Semantics, Generalized*, Proc. 1991 Intl. Logic Programming Symposium, pps 71–83, MIT Press.

[16] J. Grant, W. Litwin, N. Roussopoulos and T. Sellis. (1991) *An Algebra and Calculus for Relational Multidatabase Systems*, Proc. First International Workshop on Interoperability in Multidatabase Systems, IEEE Computer Society Press (1991) 118-124.

[17] Y. Ioannidis and T. Sellis. (1989) *Conflict Resolution of Rules Assigning Values to Virtual Attributes*, Proc. ACM SIGMOD Symp. on Management of Data.

[18] M. Kifer, G. Lausen and J. Wu. (1990) *Logical Foundations of Object-Oriented and Frame-Based Languages*, Tech. Report 90/14, SUNY at Stonybrook.

[19] M. Kifer and E. Lozinskii. (1989) *RI: A Logic for Reasoning with Inconsistency*, 4-th Symposium on Logic in Computer Science, Asilomar, CA, pp. 253-262. Full version to appear in: Journal of Automated Reasoning.

[20] M. Kifer and V.S. Subrahmanian. (1989) *Theory of Generalized Annotated Logic Programming and its Applications*, Journal of Logic Programming, 12, 4, pps 335–368, 1992. Preliminary version in: Proc. 1989 North American Conf. on Logic Programming, MIT Press.

[21] W. Kim and J. Seo. (1991) *Classifying Schematic and Data Heterogeneity in Multidatabase Systems*, IEEE Computer, Dec. 1991.

[22] R. Krishnamurthy, W. Litwin and W. Kent. (1991) *Language Features for Interoperability of Databases with Schematic Discrepancies*, Proc. ACM SIGMOD 1991.

[23] A. Lefebvre, P. Bernus and R. Topor. (1992) *Querying Heterogeneous Databases: A Case Study*, draft manuscript.

[24] J. Lu, N. Murray and E. Rosenthal. (1993) *Signed Formulas and Annotated Logics*, draft manuscript. Preliminary version in: Proceedings of the International Symposium on Multiple-Valued Logic, IEEE Computer Society Press, 1993, 48-53.

[25] A. Martelli and U. Montanari. (1982) *An Efficient Unification Algorithm*, ACM Trans. on Prog. Lang. and Systems, 4, 2, pps 258–282.

[26] R. Ramakrishnan. (1991) *Magic Templates: A Spellbinding Approach to Logic Programs*, J. of Logic Programming, 11, pps 189–216.

[27] H. Seki and H. Itoh. (1989) *A Query Evaluation Method for Stratified Programs under the Extended CWA*, Proc. 5th Intl. Conf./Symp. on Logic Programming (eds. K. Bowen and R. Kowalski), pps 195–211.

[28] H. Seki. (1989) *On the Power of Alexander Templates*, Proc. 8th ACM Symp. on Principles of Database Systems, pps 150–159.

[29] A. Sheth and J. Larson. (1990) *Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases*, ACM Computing Surveys, 22, 3, pp 183–236.

[30] J. Shoenfield. (1967) *Mathematical Logic*, Addison Wesley.

[31] A. Silberschatz, M. Stonebraker and J. D. Ullman. (1991) *Database Systems: Achievements and Opportunities*, Comm. of the ACM, 34, 10, pps 110–120.

[32] V.S. Subrahmanian. (1992) *Amalgamating Knowledge Bases*, Univ. of Maryland Tech. Report CS-TR-2949, Aug. 1992. Submitted to ACM – TODS, August 1992, revised May 1993.

[33] V.S. Subrahmanian. (1992) *Paraconsistent Disjunctive Deductive Databases*, Theoretical Computer Science, Vol. 93, pps 115–141.

[34] H. Tamaki and T. Sato. (1986) *OLD Resolution with Tabulation*, Proc. 3rd Intl. Conf. on Logic Programming (ed. E. Shapiro), pps 84–98, Springer.

[35] A. van Gelder. (1989) *The Alternating Fixpoint of Logic Programs with Negation*, Proc. 8th ACM Symp. on Principles of Database Systems, pps 1 – 10.

[36] W.K. Whang, S. B. Navathe and S. Chakravarthy. (1991) *Logic-Based Approach for Realizing a Federated Information System*, Proc. First International Workshop on Interoperability in Multidatabase Systems, IEEE Computer Society Press (1991) 92–100.

[37] , G. Wiederhold, S. Jajodia, and W. Litwin. Dealing with granularity of time in temporal databases. In *Proc. 3rd Nordic Conf. on Advanced Information Systems Engineering*, Lecture Notes in Computer Science, Vol. 498, (R. Anderson et al. eds.), Springer-Verlag, 1991, pages 124–140.

[38] G Wiederhold, S. Jajodia, and W. Litwin. Integrating temporal data in a heterogeneous environment. In *Temporal Databases*. Benjamin/Cummings, Jan 1993.

[39] R. Zicari, S. Ceri, and L. Tanca. (1991) *Interoperability between a Rule-Based Database Language and an Object-Oriented Language*, Proc. First International Workshop on Interoperability in Multidatabase Systems, IEEE Computer Society Press (1991) 125-135.

Appendix A: Proofs of Results on S-Resolution

Proof of Theorem 1. Suppose C^* and Q^* are in set expansion form as specified in Definition 5 and 8. Let θ be the mgu of A_0 and B_i.

Suppose I S-satisfies C^* and Q^*_k and $(Q^*_{k+1})\sigma$ is a ground instance of (Q^*_{k+1}). Since $Q^*_k\theta\sigma$ and $C^*\theta\sigma$ must be ground and $I \models^{\mathrm{S}} Q^*_k$,$I \models^{\mathrm{S}} C^*$ it must be the case that $I \models^{\mathrm{S}} Q^*_k\theta\sigma$ and $I \models^{\mathrm{S}} C^*\theta\sigma$. We need to show that I S-satisfies (Q^*_{k+1}). Since I S-satisfies $Q^*_k\theta\sigma$, it must S-satisfy one of the amalgamated atoms $B_j : [D_{q_j}, \mu_{q_{s_j}}]\theta\sigma$. There are two cases to consider:

- __**Case 1:**__ $(j \neq i)$ In this case, $B_j : [D_{q_j}, \mu_{q_{s_j}}]\theta\sigma$ occurs in $(Q^*_{k+1})\sigma$ and I S-satisfies this atom in $(Q^*_{k+1})\sigma$, and therefore satisfies the resolvent.
- __**Case 2:**__ $(j = i)$ In this case, I must S-satisfy $B_i : [D_{q_i}, \mu_{q_{s_i}}]\theta\sigma$ in $Q^*_k\theta\sigma$. Since I S-satisfies $C^*\theta\sigma$, there are two cases to consider:
  - __**Case 2.1:**__ I falsifies the body of $C^*\theta\sigma$. Then, there must be at least one atom A_k : $\mathtt{EXP}([D_k, \mu_k])\theta\sigma$ that is not S-satisfied in I. Let $\mu_I = \sqcup_{d \in D_k} I(A\theta\sigma)(d)$. Since $\mu_I \notin \mu_k$, it must be the case that, $\mu_I \in (\mathcal{T} \setminus \mu_k)$. Then, $A_k : \mathtt{COMP}([D_k, \mu_k])\theta\sigma$ must be S-satisfied in I. Since this atom occurs in $(Q^*_{k+1})\sigma$, I satisfies (Q^*_{k+1}).
  - __**Case 2.2:**__ I S-satisfies both the body and the head of the clause $C^*\theta\sigma$. Then, by the definition of S-satisfaction there exists a truth value $\mu' \in\Uparrow \mu_0$ such that I **A**-satisfies $A_0 : [D_0, \mu']\theta\sigma$. Then, since $D_0 \subseteq D_{q_i}$, I must **A**-satisfy an annotation $B_i : [D_{q_i}, \mu'']\theta\sigma$ such that, $\mu'' \geq \mu' \geq \mu$. This implies that, $\mu'' \in\Uparrow \mu$ and this annotation occurs in the resolvent. Therefore, I S-satisfies the resolvent. □

The proof of the Completeness Theorem (Theorem 2) for S-resolution needs several intermediate theorems that are stated below.

Theorem 5 (Ground Completeness of S-resolution) Suppose Q is the ground query $\leftarrow A : [D, \mu]$, $P \models A : [D, \mu]$, and that P possesses the fixpoint reachability property. Then, there is an *unrestricted* S-refutation of $(\leftarrow Q)^*$ from P^*.
(An unrestricted refutation does not require the unifier used at each deduction step to be the most general unifier.)

Proof: As P satisfies the fixpoint reachability property, we know that $\mathbf{A}_Q \uparrow k$ satisfies $A : [D, \mu]$ for some $k < \omega$. We proceed by induction on k.
__Base case ($k = 1$)__ According to the definition of $\mathbf{A}_Q$, there exist ground instances

$$\begin{array}{rl} A : [D_1, \mu_1] & \leftarrow \\ A : [D_2, \mu_2] & \leftarrow \\ & \cdots \\ A : [D_m, \mu_m] & \leftarrow \end{array}$$

of a finite set of clauses

$$A_1 : [D_1', \mu_1'] \leftarrow$$

$$
\begin{array}{rcl}
A_2 : [D_2^{'}, \mu_2^{'}] & \leftarrow & \\
 & \ldots & \\
A_m : [D_m^{'}, \mu_m^{'}] & \leftarrow &
\end{array}
$$

in P, $m \geq 1$, such that $\sqcup\{\mu_1, \ldots, \mu_m\} \geq \mu$ and $\bigcup_{1 \leq j \leq m} D_j \subseteq D$. Note that for all $1 \leq i \leq m$, there is a substitution θ_i, such that $A_i\theta_i = A$, $[D_i^{'}, \mu_i^{'}]\theta = [D_i, \mu_i]$. By the definition of regular representation, P^* contains ground instances

$$
\begin{array}{rcl}
A : \texttt{EXP}([D_1, \mu_1]) & \leftarrow & \\
A : \texttt{EXP}([D_2, \mu_2]) & \leftarrow & \\
 & \ldots & \\
A : \texttt{EXP}([D_m, \mu_m]) & \leftarrow &
\end{array}
$$

of unit clauses

$$
\begin{array}{rcl}
A_1 : \texttt{EXP}([D_1^{'}, \mu_1^{'}]) & \leftarrow & \\
A_2 : \texttt{EXP}([D_2^{'}, \mu_2^{'}]) & \leftarrow & \\
 & \ldots & \\
A_m : \texttt{EXP}([D_m^{'}, \mu_m^{'}]) & \leftarrow &
\end{array}
$$

and $(\leftarrow Q)^* = A : \texttt{COMP}([D, \mu]) \leftarrow$. Since for all $1 \leq i \leq m$, $D_i \subseteq D$, $(\leftarrow Q)^*$ resolves with all $A_i : \texttt{EXP}([D_i, \mu_i])$. It follows that there is an S-refutation

$\langle A : \texttt{COMP}([D, \mu]) \leftarrow, A : \texttt{EXP}([D_1, \mu_1]) \leftarrow, \theta_1\rangle$,
$\langle A : \text{S-INT}(\texttt{COMP}([D, \mu]), \texttt{EXP}([D_1, \mu_1])) \leftarrow, A : \texttt{EXP}([D_2, \mu_2]) \leftarrow, \theta_2\rangle$,
$\ldots$,
$\langle A : [D, (\mathcal{T} \setminus \Uparrow \mu) \ \cap \ \cap_{1 \leq i \leq m} \Uparrow \mu_i] \leftarrow, -, -\rangle$.

We must show that the last query evaluates to $\emptyset$. Let $\mu_{lub} = \sqcup\{\mu_1, \ldots, \mu_m\}$. Since $\mu_{lub} \geq \mu$, we have $\Uparrow \mu_{lub} \subseteq \Uparrow \mu$, hence $\Uparrow \mu_{lub} \cap (\mathcal{T} \setminus \Uparrow \mu) = \emptyset$. Then, it suffices to show that $(\cap_{1 \leq i \leq m} \Uparrow \mu_i) \subseteq \Uparrow \mu_{lub}$. For all $\mu_k \in (\cap_{1 \leq i \leq m} \Uparrow \mu_i)$, we have that $\mu_k \geq \mu_j$ for all j. Since μ_{lub} is the smallest such truth value, we must have $\mu_k \geq \mu_{lub}$ and therefore $\mu_k \in \Uparrow \mu_{lub}$.

<u>Inductive Case ($k > 1$)</u> By the definition of $\mathbf{A}_Q$, there exist ground instances $C_1\theta_1, \ldots, C_m\theta_m$ of the form

$$
\begin{array}{rcl}
A : [D_1, \mu_1] & \leftarrow & B_1^1 : [D_1^1, \mu_1^1] \& \ldots \& B_{k_1}^1 : [D_{k_1}^1, \mu_{k_1}^1] \\
A : [D_2, \mu_2] & \leftarrow & B_1^2 : [D_1^2, \mu_1^2] \& \ldots \& B_{k_2}^2 : [D_{k_2}^2, \mu_{k_2}^2] \\
 & \ldots & \\
A : [D_m, \mu_m] & \leftarrow & B_1^m : [D_1^m, \mu_1^m] \& \ldots \& B_{k_m}^m : [D_{k_m}^m, \mu_{k_m}^m]
\end{array}
$$

of clauses $C_1, \ldots, C_m$

$$
\begin{array}{rcl}
A_1 : [D_1^{'}, \mu_1^{'}] & \leftarrow & B_1^1 : [D_1^{1'}, \mu_1^{1'}] \& \ldots \& B_{k_1}^1 : [D_{k_1}^{1'}, \mu_{k_1}^{1'}] \\
A_2 : [D_2^{'}, \mu_2^{'}] & \leftarrow & B_1^2 : [D_1^{2'}, \mu_1^{2'}] \& \ldots \& B_{k_2}^2 : [D_{k_2}^{2'}, \mu_{k_2}^{2'}] \\
 & \ldots & \\
A_m : [D_m^{'}, \mu_m^{'}] & \leftarrow & B_1^m : [D_1^{m'}, \mu_1^{m'}] \& \ldots \& B_{k_m}^m : [D_{k_m}^{m'}, \mu_{k_m}^{m'}]
\end{array}
$$

in P,$m \geq 1$ such that $\sqcup\{\mu_1, \ldots, \mu_m\} \geq \mu$, $\bigcup_{1 \leq j \leq m} D_j \subseteq D$ and $\mathbf{A}_Q \uparrow (k-1) \models B_1^i : [D_1^i, \mu_1^i] \& \ldots \& B_{k_i}^i : [D_{k_i}^i, \mu_{k_i}^i]$ and there is a substitution θ_i, such that $A_i\theta_i = A$, $[D_i^{'}, \mu_i^{'}]\theta = [D_i, \mu_i]$, for all $1 \leq i \leq m$. By the definition of regular expression, P^* contains ground instances $C_1^*\theta_1, \ldots, C_m^*\theta_m$

$$\begin{array}{rcl}
A : \mathtt{EXP}([D_1, \mu_1]) & \leftarrow & B_1^1 : \mathtt{EXP}([D_1^1, \mu_1^1]) \& \ldots \& B_{k_1}^1 : \mathtt{EXP}([D_{k_1}^1, \mu_{k_1}^1]) \\
A : \mathtt{EXP}([D_2, \mu_2]) & \leftarrow & B_1^2 : \mathtt{EXP}([D_1^2, \mu_1^2]) \& \ldots \& B_{k_2}^2 : \mathtt{EXP}([D_{k_2}^2, \mu_{k_2}^2]) \\
& \ldots & \\
A : \mathtt{EXP}([D_m, \mu_m]) & \leftarrow & B_1^m : \mathtt{EXP}([D_1^m, \mu_1^m]) \& \ldots \& B_{k_m}^m : \mathtt{EXP}([D_{k_m}^m, \mu_{k_m}^m])
\end{array}$$

of clauses $C_1, \ldots, C_m$

$$\begin{array}{rcl}
A_1 : \mathtt{EXP}([D_1^{'}, \mu_1^{'}]) & \leftarrow & B_1^1 : \mathtt{EXP}([D_1^{1'}, \mu_1^{1'}]) \& \ldots \& B_{k_1}^1 : \mathtt{EXP}([D_{k_1}^{1'}, \mu_{k_1}^{1'}]) \\
A_2 : \mathtt{EXP}([D_2^{'}, \mu_2^{'}]) & \leftarrow & B_1^2 : \mathtt{EXP}([D_1^{2'}, \mu_1^{2'}]) \& \ldots \& B_{k_2}^2 : \mathtt{EXP}([D_{k_2}^{2'}, \mu_{k_2}^{2'}]) \\
& \ldots & \\
A_m : \mathtt{EXP}[(D_m^{'}, \mu_m^{'}]) & \leftarrow & B_1^m : \mathtt{EXP}([D_1^{m'}, \mu_1^{m'}]) \& \ldots \& B_{k_m}^m : \mathtt{EXP}([D_{k_m}^{m'}, \mu_{k_m}^{m'}])
\end{array}$$

By the inductive hypothesis, there is an S-refutation R_i of

$$B_1^i : \mathtt{COMP}([D_1^i, \mu_1^i]) \& \ldots \& B_{k_i}^i : \mathtt{COMP}([D_{k_i}^i, \mu_{k_i}^i]) \leftarrow$$

for all $1 \leq i \leq m$. By the same argument above, $(\mathcal{T} \setminus \Uparrow \mu) \ \cap \ \cap_{1 \leq i \leq m} \Uparrow \mu_i = \emptyset$. Therefore, $(\leftarrow Q)^*$ has an unrestricted S-refutation as follows:

$\langle A : \mathtt{COMP}([D, \mu]) \leftarrow, C_i^*, \theta_i \rangle$,
$\ldots$,
$\langle A : [D, (\mathcal{T} \setminus \Uparrow \mu) \ \cap \ \cap_{1 \leq i \leq m} \Uparrow \mu_i = \emptyset] \leftarrow, -, - \rangle$,
$R_1, \ldots, R_m$,
$\langle \leftarrow, -, - \rangle$. □

The completeness of S-resolution may now be established from the ground completeness result using standard techniques.

Lemma 2 (Mgu Lemma) Suppose there is an unrestricted S-refutation $(\leftarrow Q)^*\theta$ from an amalgamated knowledge base P. Then there is an S-refutation of $(\leftarrow Q)^*$ from P. □

Lemma 3 (Lifting Lemma) Suppose there is an S-refutation of $(\leftarrow Q)^*\theta$ from an amalgamated knowledge base P. Then there is an S-refutation of $(\leftarrow Q)^*$ from P. □

The completeness of S-resolution is an immediate consequence of the ground completeness theorem and Mgu lemma.