Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

Declarative View Updates in Linear Logic Databases

Dong-Tsan LEE C.P. TSANG
Logic and AI Lab., Department of Computer Science,
The University of Western Australia,
Nedlands, Perth, Western Australia,
Australia

Abstract

The problem of updating deductive databases through views is an important practical problem that has attracted much interest. In this paper, a systematic approach to the problem of view updating in linear logic databases is described. The semantics of views, constraints, and view updates is defined declaratively in linear logic. In contrast to classical logic, we can formalise non-shared view and transition constraints easily. An additional advantage is that the associated meaning of a given relation can be defined in terms of the validity of a legal update in a given relation. We also defined formally the update principles and showed the correctness of the update translation algorithms. In this approach, view DELETIONs are special cases of view REPLACEMENTs. This permits three transactional view update operations (INSERTION, DELETION, REPLACEMENT) in comparison to only (INSERTION, DELETION) in most exiting systems.

Keywords: Deductive Databases, View updates, Linear logic, Database updates, Knowledge representation, Views, Constraints, Declarative support.

1 Introduction

In this paper, we consider the *view update* problems in linear-logic databases. View update in databases has two main problems. The first one is how to compute the next database state with respect to updates on the underlying database. This problem has been solved by our previous research[17], which centers on the computation of next database state within first-order linear logic theory. For example, the database is initially Δ, F. Assuming that we wish to delete the fact F which is not prefixed with the exponential operator !. Rather than deleting F directly, we add to the database the formula $(F \multimap 1)$. As a consequence, we have Δ, F, $(F \multimap 1) \vdash$. Thus the next database state Σ can be deduced by proving the sequent $\Delta, F, (F \multimap 1) \vdash \Sigma$. We also provided proof search strategies to obtain the next database state Σ. These are described in our paper[17]. Our approach is also free from the AI frame problem. This is because we treat facts as resources that can be consumed and produced. Hence there is no need for the frame axioms to be stated explicitly. The second problem is how to translate a given update on a user view into an update in the database. In this paper, we solve this view update problem with respect to a linear-logic database. Many distinct advantages will be emphasised.

We also consider a linear logic database as a linear logic theory. A theory consists of a consistent set of formulas in first-order linear logic. In our framework, we use the full linear logic as defined by Girard[7, 8]. In logic databases, a database *relation* corresponds to a predicate. A *view* is associated with a *view relation* and a *view definition*[20]. A *view relation* is a rule-defined relation that is made to appear like a *base relation* to the user. A database relation defined directly by its tuples is called a *base relation*. A *rule-defined relation* is defined to be a database relation based on a set of rules. The rules that define the view relation constitute the view definition. For a view to be useful, users must be able to apply query and update operations. In general, a mapping is required to translate view updates into the corresponding updates on the underlying database. However, such a mapping does not always exist. Even when it does exist, it may not be unique because a view defines a many-to-many mapping from the base relation tuples to the view tuples[4, 20]. Thus we use a translator to decide between the alternatives.

Let us make it clear that we are concerned here with declarative support. Many approaches to a theory of updates[2, 3, 4, 6, 9, 13, 16, 23, 24] do not emphasize declarative support. Declarative support means that the operational semantics can be captured by using the same specification language as the database. In general, declarative support is better than procedural support which uses stored or triggered operations. In this paper, we define a declarative view, declarative constraints, and declarative view updates. By using declarative

views, many traditional view update problems can be overcome. One of the motivation of using views is to hide sensitive information and to control the access privileges of a user. Views provide *logical data independence* by allowing certain changes to be made to the database schema without affecting the application programs. Different users can also have different views of the same data. One problem with classical logic databases is that it cannot formalise non-shared views, declarative transition constraints and declarative (view) updates. This is because classical logic cannot handle the resource and it cannot specify database state transition. Recently, Date and McGoveran[5] described a systematic approach to the problem of updating relational views. The main contribution of [5] was the identification of a series of principles that must be satisfied by any view updating procedures. However, the approach of that paper is rather informal. It is our intention to formulate some of those update principles with our formulation of view update in linear logic databases. There has been much research of view update in deductive databases [3, 6, 9, 16]. However, they only allow DELETION or INSERTION transactional operations, because REPLACEMENT is treated as a shorthand for the DELETE-then-INSERT operation. However, this shorthand can lead to inappropriate results due to its transaction of non-ground atoms. We clearly need to support for all three update operations (DELETION, INSERTION, REPLACEMENT).

Linear Logic was introduced by Girard[7] in 1987. It is a state based resource-sensitive logic. Increasingly, computer scientists and proof theorists have recognised linear logic as an expressive and powerful logic suitable for capturing the semantics of computation. The expressive power of linear logic is evidenced by many computer science applications. Lafont has initiated some of these applications in the areas of logic programming[14] and operational semantics[15]. Martí-Oliet and Meseguer present a systematic correspondence between Petri-nets, linear logic theories, and linear categories[19]. With regard to linear logic programming languages, there have been various proposals, including Forum[22], Lygon[25], ACL[12], Lolli[11], and Linear Objects[1].

2 Preliminary

Linear logic differs from classical and intuitionistic logic in several ways. The difference is that Linear logic is a constructive logic and is suitable for the "Proof as Program" paradigm. Classical logic is not constructive due to the use of structural inference rules such as *weakening* and *contraction*. While intuitionistic logic is constructive, it is not symmetric (negation is void meaning unprovable).

Linear logic specifically does not have two structural rules, contraction and weakening. Removing the two rules gives a linear system in which each resource(formula) must be used exactly once. Once the two rules are dropped, the two possible traditions for the right $\wedge$-rule in the Gentzen classical sequent calculús,

$$\frac{\Gamma \vdash A, \Sigma_1 \quad \Delta \vdash B, \Sigma_2}{\Gamma,\Delta \vdash A \wedge B, \Sigma_1, \Sigma_2}\ (\wedge R) \qquad \text{and}$$

$$\frac{\Gamma \vdash A, \Sigma \quad \Gamma \vdash B, \Sigma}{\Gamma \vdash A \wedge B, \Sigma}\ (\wedge R^*)$$

are no longer equivalent; the removal of contraction and weakening leads to two forms of conjunctions, namely $\otimes$(times) and &(with), and similarly to two forms of disjunctions, $\wp$(par) and $\oplus$(plus). Moreover, Girard[7] discovered that in coherent spaces, the function space $A \Rightarrow B$ can be split into $!A \multimap B$, where ! is the exponential operator, and $\multimap$ is the linear implication. But linear logic is not logic without weakening and contraction. To restore the power of intuitionistic and classical logic, two modal (exponential) operators ? and ! are introduced, with contraction and weakening as their main logical rules. The main difference is that we now control in many cases the use of contraction and weakening; unlimited reuse or consumption is allowed only at formulas specifically marked with ! or ?, respectively. Girard also showed that it was possible to translate intuitionistic or classical logic into linear logic(see [7]). Details of Linear Logic can be found in references[7, 8, 18].

2.1 The Language

The language includes a set of finite terms, a countable set of predicates and a set of logical connectives. A term is defined as being a variable or a constant. Constants are usually names of objects, such as John, Lily, 3. Variable symbols are customarily lower-case unsubscripted or subscripted letters, x, y, z, t_1,.... Predicates are customarily upper case letters P, Q, ... or expressive strings of upper-case letters such as LESS and EMOTION. Each predicate has an arity. It allows to define a set of atomic formulas: if P is an n-ary predicate and t_1,..., t_n are terms. Then $P(t_1,..., t_n)$ is an atomic formula. A set of logical connectives is given as { $\otimes$, $\wp$, &, $\oplus$, $\multimap$, $\forall$, $\exists$, !, ?, 0, 1, $\bot$, $\top$ }. The connectives of linear logic are described as follows:

* Linear negation is denoted by $(.)^{\perp}$: Unlike Negation As Failure paradigm, linear negation is an involution satisfying De Morgan-like properties. Thus, it is possible to describe linear logic in terms of one-sided sequents, transforming $\Delta \vdash \Sigma$ to $\vdash \Delta^{\perp}, \Sigma$. Yet unlike classical negation, linear negation has a simple constructive meaning.
* The linear implication $-\circ$: $A -\circ B$ means "if the resource A was available, then the computation could go to state B". For example, DOLLAR(1) $-\circ$ FUJIAPPLE(3) says that with one dollar one can get three Fuji-apples. Obviously, one dollar must be used exactly once in obtaining 3 Fuji-apples.
* The multiplicative conjunction $\otimes$(times), and the dual disjunction $\wp$(par), with the neutral elements 1 and $\perp$, respectively: The tensor product $A \otimes B$ expresses the availability of the two resources A and B; both will be used. On the other hand, $A \wp B$ stands for a dependency between A and B; $A \wp B$ can either be read as $A^{\perp} -\circ B$ or as $B^{\perp} -\circ A$, i.e., "$\wp$" is a symmetric form of "$-\circ$". The constants *true* and *falsity* have their multiplicative version 1 and $\perp$, and their additive version T and 0. Also, note that $1 \otimes A = A$, and $\perp \wp A = A$.
* The additive conjunction &(with), and the dual disjunction $\oplus$(plus), with the neutral elements T and 0, respectively: A & B appears as a kind of external choice in the terminology of CSP[10]. It expresses the availability of the two types of resources A and B; only one of them will be chosen. Dually, $A \oplus B$ stands for the possibility of either A or B, but you do not know which. It appears as an internal choice. In addition, note that $T \& A = A$, and $0 \oplus A = A$.
* The exponential ! (of course), and the dual ? (why not): ! and ? are modalities. !A means the unlimited availability of the resource A. A computational metaphor often used to talk about this is the fact of storing a datum in a computer memory, where it can be read as many times as necessary. Dually, ?A denotes that A can be consumed as many times as necessary. In terms of computer, ! and ? may indicate which kind of memory operation has been performed. With this interpretation, the rules for ! and ? correspond to storing, erasing, reading and duplicating.
* The quantifier $\forall$ (every) and the dual $\exists$(some), which are pretty much the same as in classical logic.

Gentzen-stype sequent calculus rules for the Classical Linear Logic are given in Appendix A. A *sequent* is composed of two sequences of formulas separated by a $\vdash$, or turnstile symbol. The intended meaning of the sequent A1, ..., An $\vdash$ B1,...,Bm is that $A1 \otimes ... \otimes An -\circ B1 \wp ... \wp Bm$. A *sequent calculus proof rule* consists of a set of hypothesis sequents, displayed above a horizontal line, and a single conclusion sequent, displayed below the line. When investigating sequent calculus rules for linear logic, one notices that ($\otimes$R) rule and ($-\circ$L) rule require disjoint contexts whereas (&R) rule and ($\oplus$L) rule work with twice the same context.

Throughout the paper, we use the following notation conventions: Multisets of formulas will be referred by the letters Γ, Δ, Σ, ... The letters can possibly be indexed by integers. Moreover, a linear logic formula which is prefixed with the exponential operator !(of course) or ?(why not) is called *an exponential formula.* We often drop universal quantifiers in formulae. For instance, we just write $A(x) -\circ B(x)$, instead of $\forall x (A(x) -\circ B(x))$.

3 View updates

3.1 Declarative view definitions

In general, **declarative** operational semantics can be expressed in terms of the specification language. Thus these operations can be reasoned as part of the specification logic system. Linear logic provides a convenient way to formalise declarative views. We define a view in a linear logic database as a formula of the following forms, and it can be stored in the dictionary. First, a non-shared view is a non-exponential linear logic rule:

$$F -\circ L$$

, where F is a general linear logic formula and the view relation L is a literal. The non-shared view states that only one user can access the view at the given time. The ability to have non-shared views produces a number of beneficial results. In particular, we can define a view for a particular user, thus providing data security. By contrast, classical logic cannot define the non-shared view, because it cannot handle resource issues. Some examples about non-shared views can be described as follows.

1. *Join* view:

$$S(sno, sname, city) \otimes SP(sno, pno) -\circ SSP(sno, sname, city, pno).$$

Here, S(sno, sname, city) and SP(sno, pno) are database relations. SSP(sno, sname, city, pno) is a view relation.

2. *Intersection* view:

$$A(x, y) \otimes B(x, y) -\circ Q(x, y).$$

Here, A(x, y) and B(x, y) are relations, and Q(x, y) is a view relation.

3. *Union* view:

$$A(x, y) \oplus B(x, y) -\circ P(x, y).$$

Here, A(x, y) and B(x, y) are relations, and P(x, y) is a view relation

The second form is a shared view and it is an exponential linear logic rule defined as follows:

$$!(F \multimap L)$$

, where F is a general linear logic formula and the view relation L is a literal. Please note that views defined by classical logic can be transformed into shared views in linear logic.

3.2 Declarative constraints

Every one knows that data integrity is important. It is highly desirable for integrity constraints to be managed **declaratively** instead of procedurally so that constraints can be combined and reasoned easily. Our constraint checking method deals directly with query constraints of the form

$$!F.$$

Here, the constraint is expressed as a reusable formula, and F is the general linear logic formula. And, we classify integrity constraints into three kinds, namely column constraints, predicate constraints, and database constraints, as follows:

* A *column constraint* states that the value(argument) appearing in a specific database relation must be drawn from some specific domain. For example, consider the student base relation

  STUDENT(sno, sname, sdept, gpa).

  The arguments of that relation are subject to the following column constraints:

  ! INDOM(sno, SNO_DOM).
  ! INDOM(sname, SNAME_DOM).
  ! INDOM(sdept, SDEPT_DOM).
  ! INDOM(gpa, GPA_DOM).

* A *predicate constraint* states that a specific database relation must satisfy some specific condition, where the condition in question refers **solely** to the database relation under consideration — i.e., it does not refer to any other database relation, nor to any domain. For example, here is a predicate constraint for the base relation EMP(eno, ename, deptno, sal):

  $!(EQ(edept, D1) \multimap LE(sal, 50K))$

  The constraint says that employees in department D1 must have a salary less than 50K. EQ(edept, D1) expresses that dept is equal to D1. And, LE(sal, 50K) says that sal is less than 50k.

  Furthermore, we divide predicate constraints into *static* vs. *transition* constraints. Static constraints are concerned with those integrity constraints whose enforcement depends on only one state of the database. Transition constraints are those which represent dynamic properties of the database and control the proper transition between two consecutive database states. Please note that classical logic is not convenient for transition constraints[17]. For example, in situation calculus[23], a transition constraint stating that 'salaries must never decrease during the evolution of the database' can be expressed by:

  $$(\forall s,s')(\forall p,\$,\$').$$
  $$S_0 \leq s \wedge s \leq s' \wedge sal(p,\$,s) \wedge sal(p,\$',s') \supset \$ \leq \$'$$

  This representation within the situation calculus is awkward. It would be preferable to ignore the *ad hoc* temporal parameters S_0, s, s'. Using linear logic, the transition constraint can be described as follows:

  $!((SAL(p, x) \multimap SAL(p, y)) \multimap GE(y, x))$.

  Here, we encode transition constraints as reusable linear implications. SAL(p, x) expresses that the salary of the person p is x. GE(y, x) says that y is not less than x.

* A *database constraint* states that the database in question must satisfy some specific condition, where the condition in question can refer to as many database relations as desired. Furthermore, database constraints can also be divided into static vs. transition constraints. For instance, a transition constraint stating that 'When a person is fired, he is no longer an employee' can be expressed by:

  $!((EMP(x) \multimap FIRED(x)) \multimap (EMP(x) \multimap 1))$.

  Here, $EMP(x) \multimap 1$ means 'erase EMP(x)'.

3.3 View update principles

McGoveran and Date[21] tried to capture **the associated meaning** of a relation and proposed a new database design principle. We follow this important concept, and define the associated meaning in linear logic form. Every database relation certainly does have an associated meaning, and users must be aware of those meanings if they are to use the database effectively and correctly. The associated meaning of a relation constitutes the criterion for deciding whether or not some proposed update is in fact valid for the given database relation. Now it is possible to obtain a reasonably closer approximation to the associated meaning for a given database relation. We certainly know all the column constraints and predicate constraints that have been declared for a given database relation. Whenever an update is attempted on a database relation, we will verify whether or not the given update satisfies all the column constraints and predicate constraints concerned with the given database relation. Therefore, the associated meaning for a given

database relation is closely related to its column constraints and predicate constraints. Formally, we can define the associated meaning of a given base relation as follows:

Let linear logic formulae Γ represent the associated meaning of a base relation. Let Γ_1 be all column constraints that apply to that relation, and let Γ_2 be all predicate constraints that apply to that relation. Then, $\Gamma \equiv \Gamma_1 \otimes \Gamma_2$.

Since the view update problem is concerned with determining how a request to update a view can be translated into an update of the underlying base relations. We only need to consider the associated meanings of base relations. That is, it is not necessary to obtain the associated meaning of a view relation, as in Date and McGoveran[5]. They present an informal introduction to view updates on relational databases and try to compute the associated meaning for a derived table(i.e. a view relation). Formally, we state the first update principle as follows:

Assume that a view update *e* on a relation changes the current database Σ into the next database Σ'. Let Γ represent the associated meanings of the corresponding base relations. Then,

***e* is valid $\Rightarrow$ The sequent $\Sigma' \vdash \Gamma \otimes \top$ is provable**

, where the neutral element $\top$ (w.r.t. &) can erase unused resources. The principle says that if a view update *e* is valid, then the update on a relation must satisfy the associated meanings of the corresponding base relations.

3.4 Declarative view update expressions and translators

In this section, we define view update expressions in linear logic and discuss operational semantics for view updates; we present procedures which examine cut-free proof trees from a query and the linear logic database. Formally, we define a replacement expression through view with the format:

$$A(\alpha 1, ..., \alpha n) \multimap B$$

, where view $A(\alpha 1, ..., \alpha n)$ is a literal, and new data B is either a literal $A(\beta 1, ..., \beta n)$ or a neutral 1. The expression says that view $A(\alpha 1, ..., \alpha n)$ is replaced by new data B. Here we treat deletions as special cases of replacements. That is to say, if B is 1 then $A(\alpha 1, ..., \alpha n) \multimap 1$ indicates a deletion.

A REPLACEMENT TRANSLATOR

To replace $A(\alpha 1, ..., \alpha n)$ by B from a database Σ, we need to construct the cut-free proof tree for $\Sigma \vdash A(\alpha 1, ..., \alpha n) \otimes \top$, where the neutral element $\top$(w.r.t. &) can erase unused resources.

PROCEDURE for the replacement through view
Input: a linear logic database Σ, the data to be updated through view is $A(\alpha 1, ..., \alpha n)$ and let the updated version be B(i.e., B is either neutral 1 or the literal $A(\beta 1, ..., \beta n)$), and the *associated meanings* of base relations
Output: a set of replacements
BEGIN
1. IF $\Sigma \vdash A(\alpha 1, ..., \alpha n) \otimes \top$ is not provable,
   THEN exit
2. Construct all possible cut-free proof trees for
   $\Sigma \vdash A(\alpha 1, ..., \alpha n) \otimes \top$
3. Compute sets of replacements:
   Examine each proof tree to select the base relations appearing on the axioms(non-failed branches) of the proof tree, and construct the members of a replacement set. We only select the base relation which satisfies the condition: one of its arguments is connected with argument αi in $A(\alpha 1, ..., \alpha n)$, where if B is a literal then $\alpha i \neq \beta i$.
4. Choose a set:
   Check whether or not the set of replacements satisfy the update principles.
END

We have to show the correctness of the replacement translator. Correctness is expressed by the following theorem:

Theorem 1 (Correctness of Replacement Translator)
Let Σ be a linear logic database. The data to be updated through view is $A(\alpha 1, ..., \alpha n)$ and let the updated version be B(i.e. B is either 1 or $A(\beta 1, ..., \beta n)$). Let the replacement set F be one of the output of the replacement translator for the input Σ, $A(\alpha 1, ..., \alpha n)$, B, and the *associated meanings* of base relations.
Then, $\Delta \vdash A(\alpha 1, ..., \alpha n) \otimes \top$ is not provable and $\Delta \vdash B \otimes \top$ is provable, where F updates Σ to Δ.

Proof

Since we examine the proof tree for $\Sigma \vdash A(\alpha 1, ..., \alpha n) \otimes \top$ and select the base relation satisfying the following condition: one of base-relation's arguments is connected with argument αi in $A(\alpha 1, ..., \alpha n)$, we can update Σ to Δ by the replacement set F. We must prune the non-failed branches(the axioms) of the proof tree. It follows that we have each proof tree for $\Delta \vdash A(\alpha 1, ..., \alpha n) \otimes \top$ fail. That is to say, $\Delta \vdash A(\alpha 1, ..., \alpha n) \otimes \top$ is not provable. Furthermore, if B is 1, then $\Delta \vdash B \otimes \top$ always succeeds. If B is a literal, then we can construct

from the proof tree for $\Sigma \vdash A(\alpha 1, ..., \alpha n) \otimes T$ a succeeded proof tree as follows. By means of the replacement set F, the selected base-relations are replaced by new base-relations. And, $A(\alpha 1, ..., \alpha n)$ in the proof tree for $\Sigma \vdash A(\alpha 1, ..., \alpha n) \otimes T$ should be replaced by $A(\beta 1, ..., \beta n)$. Obviously, the succeeded proof tree corresponds to the proof tree for $\Delta \vdash A(\beta 1, ..., \beta n) \otimes T$. Then, $\Delta \vdash B \otimes T$ is provable..

Last, we define an insertion expression through view with the format:

$$1 \multimap A$$

, where A is a literal. The expression states that we wish to insert A into the database.

AN INSERTION TRANSLATOR

To insert a literal A through view into a database Σ, we need to construct the cut-free deduction tree for $\Sigma \vdash A \otimes T$.

PROCEDURE for the insertion view update
Input: a linear logic database Σ,
a literal A appearing in the insertion view update, and the associated meanings of base relations
Output: a set of insertions
BEGIN
1. IF $\Sigma \vdash A \otimes T$ is provable, THEN exit
2. Construct a failed deduction tree τ for $\Sigma \vdash A \otimes T$
3. Compute sets of insertions:
Examine the deduction tree τ to select the base relations as the members of an insertion set, by means of having the failed branches of the deduction tree τ succeed.
4. Choose a set:
Check whether or not the set of insertions satisfy the update principles.
IF the insertion sets are not valid, THEN
IF possible to construct another deduction tree,
THEN go to step 2
ELSE exit
END

We show below that the insertion translator is correct as expressed by theorem 2.

Theorem 2 (Correctness of the Insertion Translator)
Let Σ be a linear logic database and A a literal. Let the insertion set F be one of the output of the insertion translator for the input Σ, A, and the associated meanings of base relations.
Then, $\Delta \vdash A \otimes T$ is provable, where F updates Σ to Δ.

Proof

The proof is straightforward since the insertion set F is obtained by means of having the failed branches of the deduction tree succeed. And, F updates Σ to Δ. That is, the proof search for the linear sequent $\Delta \vdash A \otimes T$ does succeed.

4 Examples

Consider the linear logic deductive database Δ below.

STUDENT(94061, Ted, CS, 5).
STUDENT(94067, Lily, EE, 6).
STUDENT(95002, Smith, EC, 6).
STUDENT(96005, Tom, CS, 5).
STUDENT(96015, Don, EC, 7).

Let (projection) view SC be defined as a non-shared view: STUDENT(sno, sname, sdept, gpa) $\multimap$ SC(sno, gpa).
The formal meaning of base relation STUDENT is the following:
! INDOM(sno, SNO_DOM).
! INDOM(sname, SNAME_DOM).
! INDOM(sdept, SDEPT_DOM).
! INDOM(gpa, GPA_DOM).
!(EQ(sdept, CS1) $\multimap$ GT(gpa, 4)).
!(EQ(sno1, sno2) $\multimap$ (EQ(sname1, sname2) & EQ(sdept1, sdept2) & EQ(sgpa1, sgpa2))).
These linear logic expressions correspond to the following statement: "The student with the specified student number(sno) has the specified name(sname), studies in the specified department(sdept), and has the GPA(gpa). Furthermore, if the department number is CS1, then the GPA must be greater than 4. Also, no two students have the same student number."
* An attempt to insert the data SC(96017, 5) will succeed. Figure 1 shows a failed deduction tree for $\Delta \vdash$ SC(96017, 5) $\otimes$ T. Δ is split into Δ_2 and the formula STUDENT(sno, sname, sdept, gpa) $\multimap$ SC(sno, gpa) in this deduction tree. In this tree, there is a failed branch $\vdash$ STUDENT(sno, sname, sdept, gpa). The view update will have the effect of inserting the data STUDENT(96017, *n*, *d*, 5) into Δ, where *n* and *d* are variables. And the data STUDENT(96017, *n*, *d*, 5) satisfies the associated meaning for STUDENT.
* An attempt to insert the data SC(94061, 5) will fail, because it violates the associated meaning of base relation STUDENT. That is, we cannot insert duplicate data into the database. Specially, it violates the predicate constraint: !(EQ(sno1, sno2) $\multimap$ (EQ(sname1, sname2) & EQ(sdept1, sdept2) & EQ(sgpa1, sgpa2))).
* An attempt to replace SC(94061, 5) by SC(94061, 7) will succeed; the effect will be to replace the data

STUDENT(94061, Ted, CS, 5) by the data STUDENT(94061, Ted, CS, 7) — not by the data STUDENT(94061, n, d, 7). Please observe that if REPLACEMENT is regarded as shorthand for a DELETE-then-INSERT sequence, then the effect will be to replace the data STUDENT(94061, Ted, CS, 5) by the data STUDENT(94061, n, d, 7).

$$\dfrac{\dfrac{\dfrac{fail}{\vdash \text{STUDENT(sno, sname, sdept, gpa)}} \quad \dfrac{\{\text{sno}/96017, \text{gpa}/5\}}{\text{SC(sno,gpa)} \vdash \text{SC}(96017,5)}(Id)}{\text{STUDENT(sno, sname, sdept, gpa)} \multimap \text{SC(sno, gpa)} \vdash \text{SC}(96017, 5)}(\multimap L) \quad \dfrac{}{\Delta_2 \vdash \top}(\top)}{\Delta \vdash \text{SC}(96017, 5) \otimes \top}(\otimes R)$$

Figure 1. A failed deduction tree for $\Delta \vdash \text{SC}(96017, 5) \otimes \top$.

5 Discussion

In this paper, we have studied the problem of view updating based on modelling of database using linear logic. The advantages of declarative support have been emphasised. This is crucial to the successes of an intelligent database system. We have shown that the application of linear logic to databases can overcome many classical view update problems. These definitions are concise and formal. They include non-shared views, declarative transition constraints, and declarative (view) update expressions. A pleasant consequence of our approach is that deletions and replacements through views not only have the same expressions, but also the same operational semantics. We also show, through different examples, how the replacement and insertion translators produce correct translations for view updates. In general, the number of translations of a view update is exponential in the length of the database. It is crucial to use the *associated meaning* of a given database relation to reduce the number of translations of a view update. While it may appear that computational efficiency may be low, linear logic is inherently a constructive logic and efficient programs can be synthesised if optimisation processes are applied.

This research can be extended in various directions. In this paper we only consider primitive view update. The ability to define complex view updates in terms of primitive ones is extremely important for a theory of updates. One research direction is to define complex view update expression based on linear logic connectives, and to generalise the translation procedures so that we can deal with complex view updates. The problem of choosing among several alternative updates sequences that may be available for performing a view update still exists. Further research will also be focused on the different criteria which can help us to resolve ambiguity when translating view updates.

References

[1] J.M. Andreoli and R. Pareschi, Linear Objects: Logical Processes with Built-In Inheritance, *New Generation Computing.* 9 (1991) 445-473.

[2] S, Abiteboul and V. Vianu, Procedural Languages for Database Queries and Updates, *J. Comput. Syst. Sci.* 41(2)(1990)181-229.

[3] P. Atzeni and R. Torlone, Updating Datalog Databases, in: *Next Generation Information Systems Technology*, Kiew, Soviet Union, LNCS 504 (Springer-Verlag, Berlin, 1990) 347-362.

[4] E. F. Codd, Recent Investigations in Relational Data Base Systems, in: *Information Processing 74* (North-Holland Publishing Company, 1974) 1017-1021.

[5] C.J. Date and D. McGoveran, Updating Union, Intersection, and Difference Views, in: C.J.Date, ed., *Relational Database Writings, 1991-1994* (Addison-Wesley, Reading, MA, 1995).

[6] R. Fagin, G. M. Kuper, J. D. Ullman, and M. Y. Vardi, Updating Logical Databases, in: *Advances in Computing Research*, V3, (JAI Press Inc., 1986) 1-18.

[7] J. Y. Girard, Linear Logic, *Theoretical Computer Science.* 50 (1987) 1-102.

[8] J.Y. Girard: Linear Logic, Its Syntax and Semantics, in: J.-Y. Girard and Y. Lafont and L. Regnier, eds., *Advances in Linear Logic* (Cambridge University Press, 1995).

[9] A. Guessoum and J.W. Lloyd, Updating knowledge bases. *New Generation Computing* 8(1) (1990) 71-89.

[10] C.A. R. Hoare, *Communicating Sequential Processes* (Prentice-Hall, 1985).

[11] J.S. Hodas, Lolli, An Extension of λProlog with Linear Context Management, in: D. Miller, ed., *Workshop on the λProlog Programming Language*, pages 159-168, Philadelphia, Pennsylvania, August 1992.

[12] N. Kobayashi and A. Yonezawa, ACL: A Concurrent Linear Logic Programming Paradigm, in: D. Miller, ed., *Proceedings of the International Symposium on Logic Programming*, Vancouver, Canada, October 1993 (MIT Press, 1993)279-294.

[13] R. Kowalski, Databases Updates in the Event Calculus, *The Journal of Logic Programming*. 12 (1992) 121-146.

[14] Y. Lafont, Linear Logic Programming, in: *Workshop on Programming Logic*, Göteborg, (1987) 209-220.

[15] Y. Lafont, The Linear Abstract Machine, *Theoretical Computer Science*. 59(1988) 157-180 Some corrections in Volume 62 (1988) 327-328.

[16] D. Laurent, V. P. Luong, and N. Spyratos, in: *Database Updating Revisited. DOOD'93*, Arizona, USA, LNCS 760 (Springer-Verlag, Berlin, 1993).

[17] D. T. LEE and C.P. Tsang, Solving the Database Update Problem Using Linear Logic, in: R. Topor, ed., Australian Computer Science Communications. 18(2)(1996) 131-138.

[18] P. Lincoln, Linear Logic, *ACM SIGACT Notices*. 23(2) (1992) 29-37.

[19] N. Martí-Oliet and J. Meseguer, From Petri Nets to Linear Logic through Categories: A Survey, *Journal on Foundations of Computer Science*. 2(4)(1991)297-399.

[20] S. Manchanda, and D. S. A. Warren, Logic-based Language for Database Updates, in: J. Minker, ed., *Foundations of Deductive Databases and Logic Programming* (Morgan Kauffman, Los Altos, 1988) 363-394.

[21] D. McGoveran and C.J. Date, A New Database Design Principle, in: C.J.Date, ed., *Relational Database Writings, 1991-1994* (Addison-Wesley, Reading, MA, 1995).

[22] D. Miller, A Multiple-Conclusion Meta-Logic, in: *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, Paris, France, 4-7 July 1994 (IEEE Computer Society Press, 1994)272-281. *Theoretical Computer Science*, to appear.

[23] R. Reiter, On Specifying Database Updates, *The Journal of Logic Programming*. 25(1)(1995) 53-91.

[24] E. Teniente and A. Olivé, The Events Method for View Updating, in: *Deductive Databases. EDBT'92*, LNCS 580 (Springer-Verlag, Berlin, 1992).

[25] M. Winikoff and J. Harland, Implementation and Development Issues for the Linear Logic Programming Language *Lygon*, in: *Proceedings of the Eighteenth Australian Computer Science Conference*, Adelaide, Australia, 1995.

Appendix A The sequent calculus for linear logic

$$\frac{}{A \vdash A}\ (\text{Id}) \qquad \frac{\Gamma \vdash A, \Sigma_1 \quad \Delta, A \vdash B, \Sigma_2}{\Gamma, \Delta \vdash B, \Sigma_1, \Sigma_2}\ (\text{Cut})$$

$$\frac{\Gamma, A, B, \Delta \vdash \Sigma}{\Gamma, B, A, \Delta \vdash \Sigma}\ (\text{Exch. L}) \qquad \frac{\Gamma \vdash \Delta, A, B, \Sigma}{\Gamma \vdash \Delta, B, A, \Sigma}\ (\text{Exch. R})$$

$$\frac{}{\vdash 1}\ (1\text{R}) \qquad \frac{\Gamma \vdash \Delta}{\Gamma, 1 \vdash \Delta}\ (1\text{L})$$

$$\frac{\Gamma \vdash A, \Sigma_1 \quad \Delta \vdash B, \Sigma_2}{\Gamma, \Delta \vdash A \otimes B, \Sigma_1, \Sigma_2}\ (\otimes\text{R}) \qquad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \otimes B \vdash \Delta}\ (\otimes\text{L})$$

$$\frac{}{\Gamma \vdash \top, \Delta}\ (\top) \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash A \& B, \Delta}\ (\&\text{R})$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \& B \vdash \Delta}\ (\&\text{L}) \qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \& B \vdash \Delta}\ (\&\text{L})$$

$$\frac{}{\Gamma, 0 \vdash \Delta}\ (0) \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \oplus B, \Delta}\ (\oplus\text{R})$$

$$\frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \oplus B, \Delta}\ (\oplus\text{R}) \qquad \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \oplus B \vdash \Delta}\ (\oplus\text{L})$$

$$\bot \vdash\ (\bot\ \text{Left}) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \bot, \Delta}\ (\bot\ \text{Right})$$

$$\frac{\Gamma, A \vdash \Delta_1 \quad \Sigma, B \vdash \Delta_2}{\Gamma, \Sigma, A \wp B \vdash \Delta_1, \Delta_2}\ (\wp\text{L}) \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \wp B, \Delta}\ (\wp\text{R})$$

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \multimap B, \Delta}\ (\multimap\text{R}) \qquad \frac{\Gamma \vdash A, \Sigma_1 \quad \Delta, B \vdash \Sigma_2}{\Gamma, \Delta, A \multimap B \vdash \Sigma_1, \Sigma_2}\ (\multimap\text{L})$$

$$\frac{!\Gamma \vdash A, ?\Delta}{!\Gamma \vdash !A, ?\Delta}\ (!\text{S}) \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta}\ (!\text{D})$$

$$\frac{\Gamma, !A, !A \vdash \Delta}{\Gamma, !A \vdash \Delta}\ (!\text{C}) \qquad \frac{\Gamma \vdash \Delta}{\Gamma, !A \vdash \Delta}\ (!\text{W})$$

$$\frac{!\Gamma, A \vdash ?\Delta}{!\Gamma, ?A \vdash ?\Delta}\ (?\text{S}) \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash ?A, \Delta}\ (?\text{D})$$

$$\frac{\Gamma \vdash ?A, ?A, \Delta}{\Gamma \vdash ?A, \Delta}\ (?\text{C}) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash ?A, \Delta}\ (?\text{W})$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, A^{\bot} \vdash \Delta}\ (^{\bot}\ \text{Left}) \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash A^{\bot}, \Delta}\ (^{\bot}\ \text{Right})$$

$$\frac{\Gamma \vdash A[y/x], \Delta}{\Gamma \vdash \forall x\, A, \Delta}\ (\forall\, \text{R}) \qquad \frac{\Gamma, A[\eta/x] \vdash \Delta}{\Gamma, \forall x\, A \vdash \Delta}\ (\forall\, \text{L})$$

$$\frac{\Gamma \vdash A[\eta/x], \Delta}{\Gamma \vdash \exists x\, A, \Delta}\ (\exists\, \text{R}) \qquad \frac{\Gamma, A[y/x] \vdash \Delta}{\Gamma, \exists x\, A \vdash \Delta}\ (\exists\, \text{L})$$

where in (∀R) and (∃L), y does not occur free in the conclusion.