Compiling Integrity Checking into Update Procedures

Mark Wallace
ECRC, Arabellastr 17
8000 Munchen 81, Germany
mark@ecrc.de

Abstract

Integrity checking has been investigated extensively in the field of deductive databases. Methods have been developed to optimise the checking of an update by specialising the constraints for the information that could have been affected by it. The optimisation has been applied to sets of updates resulting from the execution of unspecified update procedures. This paper investigates the compilation of integrity checking into the procedures themselves. The paper introduces a (procedural) update language, and describes how constraints are compiled into procedures expressed in this language. The compilation yields conditions on the *original* database state that guarantee safety of the update. The paper also shows why compilation into procedures offers important possibilities for optimisation not available in the earlier framework.

1 Introduction

Integrity constraint enforcement is important for preventing incorrect data being entered in data and knowledge bases. In databases only quite simple constraints are generally enforced by the system, such as types and functional dependencies. However most applications require more complex constraints.

An integrity enforcement technique has been developed which separates the integrity constraints from the update procedures. First proposed in the context of relational databases [Nic82] the technique has been extensively researched in the context of deductive databases [SK88; Dec86; LST87; BDM88] and the references in [BMM90].

Under this approach the integrity constraints are expressed as logical statements - effectively yes/no queries - and stored in the knowledge base. The system just checks the constraints that could have been affected by the update, and even then only what has (or could have) changed. We call this *specialised integrity checking.* It will be described in more detail in the next section.

To date integrity specialisation has been applied for updated literals (single inserts and deletes), for sets of updated literals, and for intentional updates (inserts/deletes satisfying a certain condition). However it is normally assumed that the set of updates is the result of an update procedure which is not visible to the specialiser. Specialisation is applied to the resulting update, not to the update procedure.

In this paper we show how to specialise integrity constraints for the update procedures themselves. This yields significant reductions in the amount of checking that has to be done at update time. The reason is that update procedures often perform partial integrity enforcement automatically. For example the procedure for hiring an employee will require that he has a salary which is a real number, and that he belongs to a department etc.

In this paper we show how to compile integrity checking into update procedures in such a way that redundant checking is minimised. The aim is to eliminate any further checking at update time of constraints which are, from the design of the update procedure, bound to be satisfied after the update. The idea of compiling constraints into update procedures was first suggested in [Sto75], however the techniques applied here are quite different. Effectively we perform specialised integrity checking for update procedures at compile time.

In section 2 we study integrity checking methods in deductive databases. The next section describes the update language used for encoding update procedures. In section 4 we explain how integrity constraints are compiled into these procedures.

Specialised integrity checking implies a requirement, for reasoning over multiple database states. However it is shown that this requirement can be compiled away so that there is no need for such a facility at update time. In fact the result of compilation is a safe update procedure comprising a condition, which is simply a query against the current database state, and then the original update procedure.

2 Constraints on Deductive Databases

2.1 Constraints in Deductive Databases

Deductive databases extend relational databases by supporting *intentional* data and logical dependencies amongst the data. A deductive database is a set of program clauses, divided into facts and rules. A program clause is a formula *Head* $\leftarrow$ *Body,* with a "head" *(Head)* and a "body" *(Body).* The head is an atom, comprising predicate and arguments. The body is a conjunction of literals. A fact is a clause with an empty body. A goal is a clause with an empty head. The goal <— *G* expresses the query *G.* We use the logic programming syntax for predicates, functions and variables, eg:

grandparent(fred, Y) $\leftarrow$ *parent(fred, X),parent(X, Y)*

We will not exclude compound terms from our rules. In other words our underlying relational model need not be in first normal form. We assume, however, that appropriate syntactic restrictions on clauses are enforced to ensure completeness and termination of query evaluation. Such restrictions are described in [RBS87], for example.

The rules yield a notion of dependency:

Definition 1 *An atom A "directly depends" on a literal U if A is the head of an instantiated database rule whose body contains U or its negation. The negated literal* $\neg A$ *directly depends on U if A does.*

We assume that for each constraint, *Q,* there is a rule *inconsistent* $\leftarrow \neg Q$ in the ruleset. Therefore a knowledge base is inconsistent if and only if *inconsistent* is derivable as a consequence.

In this paper we do not consider rule updates, therefore a database state comprises a fixed set of rules and an updateable set of "base" facts. We assume, without loss of generality, that the clauses defining a database predicate are either all base facts, or else they are all rules. Predicates defined by base facts are called "base" predicates, and those defined by rules are "derived" predicates. As the result of an update certain new base facts are added to the database, and certain base facts are dropped. These are termed the "directly" updated literals, or "direct updates". Also as a result of an update certain new facts are derivable, and certain previously derivable facts are no longer derivable. These are termed the "indirectly" updated literals, or "indirect updates".

2.2 Integrity Constraint Specialisation

The derived literals affected by an update are often assumed to be included in those dependent on the direct updates.

Definition 2 *A dependent update is an indirect update that directly depends either on a direct update, or on another dependent update.*

Efficient integrity checking methods in deductive databases (see references above) depend on restricting proofs of inconsistency to those involving updated literals. These methods essentially prove that *inconsistent* is a dependent update, by searching for literals on the dependency path between the direct updates and *inconsistent,* and showing that some, or all, of them are dependently updated. Two such methods [SK88; LST87] are proved correct. The former is proved only for the special case of *positive* databases. We now introduce a basic result on which all the above methods depend. (The proof is in [Wal90].)

Theorem 1 *In a stratified deductive database, the indirect updates are all dependently updated.*

To determine whether a literal has really been updated, it is necesssary to check its truth before the update. However some integrity checking methods drop this check. Accordingly we define the class of "update consequences". First, a preliminary definition:

Definition 3 *Potential updates are literals dependent on the direct updates.*

Definition 4 *Update consequences are potential updates provable in the updated database*

By theorem 1 it follows that:
L is an update consequence, *L* was not provable in the original database $\Leftrightarrow$ *L* is a dependent update.

The update consequences are easier to calculate than the dependent updates, though they are also a larger class. Integrity checking in deductive databases can be implemented by checking if *inconsistent* belongs to the class of update consequences. The method of [LST87] effectively uses this approach.

A specification of different classes of literals affected by an update, and their application to integrity checking, is presented in [Kue90]. A recursive definition of the class of dependent updates as a logic program was given in [BDM88]. In this paper the class of literals affected by an update procedure will similarly be specified as a logic program.

3 Updates

3.1 A Procedural Update Language

We now present a procedural language for updates which enables us to define update procedures. This language offers the update facilities provided in relational database systems, but allows them to be combined freely. The standard primitive updates are provided. Then compound updates can be constructed by placing updates in sequence, applying them conditionally, or applying them to all values satisfying a query. Finally update procedure calls can also be incorporated into compound updates. The following is an abstract syntax for the language (rather than a proposed concrete syntax).

The primitive updates are as follows
insert (Rel Name, Tuple)
delete(Rel Name, Tuple)
replace(Re/Name, *Tuplel,Tuple2)*

Compound updates are constructed according to the following syntax:

```
Update ::- Primitive Update
           U_1 then U_2
           if Condition then Update
           foreach Vars.Condition do Update
           Update Procedure Call
```

Informally, the update U_1 then U_2 is a sequential update - do update U_1 followed by U_2. The update if *Condition* then *Update* is a conditional update - if the condition holds, perform the update. The condition and the update have no free variables.

The update foreach *Vars .Condition* do *Update* is a set-oriented update - for each answer satisfying the condition, do the update. *Vars* is a list of variables occurring free in *Condition.* The "answers" are values for these variables which make the condition true. In this case *Update* is an update *template* whose free variables are included in the list *Vars.* In fact conditional updates are merely a special case of set-oriented updates: where the list *Vars* of free variables is empty.

Update templates are also used in the definition of update procedures. An update procedure *Name* is defined thus:

Name(X_1,......X_n) := *Update*

where X_1,. . ., X_n are the free variables in the update template *Update.* *A* procedure for raising employees' salaries by a certain percentage can be defined as follows:

raiseSal(R) : =
  foreach *Emp, Sal, R*, *N Sal:esr(Emp, Sal, R, NSal)*
    do *replace(empSal*, *(Emp, Sal), (Emp, NSal))*

where the predicate *esr* is defined by the clause

$esr(Emp, Sal, R, NSal) \leftarrow$
  $empSal(Emp, Sal), NSal$ **is** $Sal + Sal * (R/100)$

Update procedures cannot appear directly or indirectly in their own definitions: i.e. they cannot be defined recursively. However the rules defining update conditions can involve recursion, and many "recursive" updates can be expressed just using the recursion in the conditions. An example of a recursively defined update from [Abi88] is the recursive addition of new tuples to a binary relation *R* until its transitive closure had been generated. The same effect can be achieved in our update language by defining transitive closure as a derived relation *TR,* and then inserting to *R* all tuples satisfying *TR:*

foreach *X, Y* : *tr(X, Y)* do insert(r, *(X, Y))*

A useful consequence of banning recursively defined updates is that all update procedures reduce to compound updates involving a fixed finite sequence of constructors. In particular only a fixed number of sequencing constructors are used in any update, which means that the update only involves a fixed number of intermediate states.

3.2 Semantics

To denote sequences of variables $X_1, \ldots, X_k$ and $Y_1, \ldots, Y_k$ etc. we shall henceforth use bold characters: X and Y. Also X = T will denote the conjunction of equations $(X_1 = t_1 \wedge \ldots \wedge X_n = t_n)$.

An update *U* is a function from states to states. For the state which results from applying update *U* to state *S* we write *up (U,S).* The state which results from applying the primitive update insert(p, T) to a state *S* is therefore *up(insert(p,* T), *S).* Similarly for delete.

The semantics of a sequenced update U_1 then *U2* is given by functional composition. Thus the result of applying the above sequenced update to the state *S* is *up*$(U2,$ *up*$(U1, S))$.

The result of applying the primitive update replace(p, T1, T2) to a state *S* is up(insert(p, T2), up(delete(p, T1), *S)).*

The constructor foreach builds a set of "simultaneous" updates from an update template and a condition. Informally the set-oriented update foreach X : *Cond* do *Up* denotes the set of updates *{Up(X) :Cond[X]}.*

The power and interest of this update language lies in updates which combine both set-oriented and sequenced components. Sets and sequences do not naturally combine together. Informally, it is not obvious what update would be denoted by the following set:

{(insert *p(a)* then delete *p(b)),*
  (insert *p(b)* then delete *p(a))}*

We choose, in our semantics, to assign the following meaning to this example:

{insert *p(a),* insert *p(b)}* then
  { deletep(b), delete *p(a)}*

Accordingly we treat updates of the form foreach X : *Cond* do *(U 1* then *U2)* as the following

(foreach X : *Cond* do U_1) then
  (foreach X : *Cond* do U2)

However we must be very careful about the state in which the set-oriented condition *Cond* is evaluated. For example the two following queries have a quite different semantics: "For each employee *E,* delete *employee(E)* and then insert *manager(E)",* and "For each employee *E* delete *employee(E),* then for each employee *E,* insert *manager(E)".* After the second update there are no new managers!

We temporarily introduce for each n-ary predicate *pred* in the deductive database a $n + 1$-ary predicate *predSern,* with one extra argument denoting the state. We use a shorthand for literals involving the new predicates: if *A* denotes the atom p(X), then *ASem(S)* denotes p(X, *S);* this shorthand is also extended to formulae.

We can now specify the semantics of set-oriented updates recursively. If *U* is a primitive update function (insert, or delete), then the result of applying the update foreach X : *Cond* do *U* to the state *S* is

up(foreach *X.:CondSem(S)* do *U,S).*

If *Cond* is the conjunction *Condi* ∧ *Cond2,* then the result of applying the update

foreach X : *Cond1* do foreach Y : *Cond2* do *U*

to the state *S* is

up(foreach X, Y : *CondSem(S)* do *U, S).*

Lastly, suppose *U* is a sequenced update which maps the state *S* to the state *up(U2, up(U1, S)).* Then the update foreach X : *Cond* do *U* maps the state *S* to the state

$up($ **foreach X** : $CondSem(S)$ **do** $U2,$
  $up($ **foreach X** : $CondSem(S)$ **do** $U1, S))$.

4 Compiling Constraints Into Update Procedures

In this section we draw together the two threads of integrity specialisation and update semantics.

In the original framework of integrity specialisation, it was necessary to check if $\neg$***inconsistent*** was an "update consequence". In the new framework the object of specialisation is not an update but an update procedure which maps an unspecified state *S* to a new state, which can be specified as a function of *S.*

If an update maps *S* to *NewS,* we define an update consequence to be any literal *L* for which *LSem(NewS)* holds but not *LSem(S).* Showing that *inconsistent* is not an update consequence amounts to proving that *inconsistent Strn(N ewS)* ⟶ *inconsistent Sern(S).*

In this section we describe predicates *predSem, predConsq* and *predNegConsq* which effectively axiomatise update consequences. We then show how integrity constraint compilation into update procedures is effected by partially evaluating the definition of *inconsistentConsq.* Finally we explain how this partial evaluation eliminates all new predicates *prcdSem, predConsq* and *predN egConsq* from the resulting code, so all that remains is a check on the original database state.

4.1 The drawbacks of a direct implementation

The system could simply perform the update and do integrity specialisation on the result, using the methods of section 2 above. However such a direct implementation can be unnecessarily inefficient. For example, the update foreach *Y* : *r(Y)* do (insert *p(Y)* then delete *q(Y))* cannot cause a violation of the constraint
inconsistent $\leftarrow p(Y), q(Y)$.
A direct implementation of integrity checking on this update would perform much redundant work. Every single value inserted for *p* would be checked, at update time.

The fact that this update cannot cause a violation of the constraint is a matter of logical proof. Such an impossibility can be detected using logical optimisation, as we shall show below. The compilation of integrity constraints into update procedures using logical optimisation ensures that, for the above example, no checking takes place at update time - at least not for the given constraint.

4.2 Rules Derived from the Semantics of Updates

We now give the definition for the new predicate *predSem* temporarily introduced for the denotation of each database predicate *pred* in a given state.

If *pred* is a derived predicate, then *predSem* has precisely the same definition as *pred,* except that in its body each goal *G* is replaced by the goal *GSem(S).* For example the above rule for *inconsistent* yields the following rule for *inconsistent Sent:*
inconsistent $Sern(S) \leftarrow pSem(Y,S), qSem(Y,S)$.

If *pred* is a base predicate, then for the current state (say *statel) predSem* has the same extension as *pred.* This can be expressed using a single rule:
$\mathbf{(R1)}\ predSem(\mathbf{X}, statel) \leftarrow pred(\mathbf{X}).$

However there are further rules defining the effect of each update on the base predicate. The axioms specifying the effect of primitive insertions are captured by the following rules.

$\mathbf{(R2)}\quad pSem(\mathbf{X}, up(insert(p, \mathbf{X}), S)) \leftarrow$
$\mathbf{(R3)}\quad pSem(\mathbf{X}, up(U, S)) \leftarrow$
$\qquad\qquad primitive(U), pSem(\mathbf{X}, S),$
$\qquad\qquad \neg U = \mathbf{delete}(p, \mathbf{X})$

The rules defining the set-oriented updates use higher order predicates *rename* and *call* in the rule bodies. Set-oriented insertions and deletions are defined by the following rules:

$\mathbf{(R4)}\ pSem(\mathbf{T}, up(\mathbf{foreach}\ \mathbf{X} : Cond\ \mathbf{do}\ Up, S)) \leftarrow$
$\qquad\qquad \exists\mathbf{X}.(Up = \mathbf{insert}(p, \mathbf{T}) \wedge call(Cond))$
$\mathbf{(R5)}\ pSem(\mathbf{T}, up(\mathbf{foreach}\ \mathbf{X} : Cond\ \mathbf{do}\ Up, S)) \leftarrow$
$\qquad\qquad pSem(\mathbf{T}, S),$
$\qquad\qquad \neg\exists\mathbf{X}.(Up = \mathbf{delete}(p, \mathbf{T}) \wedge call(Cond))$

As explained in section 3 above, the update *Up* is always primitive. Given an update, it is always possible to eliminate the meta-predicate *call* by partial evaluation. Finally the existentially quantified goals can be translated to atomic goals as described in section 4.4.2 below.

Sequenced updates are automatically dealt with by rules R2-R5. In this case the state *S* is itself a compound term of the form *up*(U, S').

As a short example, assume *r* is defined by the rule
$r(X,Y) \leftarrow p(X), q(Y)$
and consider the query "does r(a,b) hold after an update insert(p,X)?".

The goal *rSern(a, b,* up(insert(p, *X), statel))* can be reduced using the rule
$rSem(X,Y,S) \leftarrow pSem(X,S), qSem(Y,S)$
to the pair of goals *pSem(a,* up(insert(p, *X), state1))* and $qSem(b, up(\mathbf{insert}(p, X), state1))$. By rule R2, the goal for *pSem* is satisfied if A' = a, otherwise it can be reduced using the rule R3 to to *pSem(a, statel).* This can be reduced to *p(a)* using R l . The goal for *qSem* reduces to *q(b).*

We conclude that the query is satisfied if *q(b)* holds (in the original state), and if either $p(a)$ holds or if $X = a$. Note that this conclusion does not mention *statel.*

4.3 How to Check Only What Has Changed

Just as it is possible to give rules (for *predSem)* for deriving what is true (for *pred)* in an updated state, it is possible to give rules deriving what has changed between two states. Thus we can derive the update consequences for a given update. The predicate which captures the positive update consequences for *pred* is named *predConsq.* Another predicate captures the negative update consequences (those facts for *pred* which become false in the new state). It is named *predN egConsq.*

These predicates are defined by rules quite similar to those for *predSem.* For example if *pred* is a base predicate, then the rules dealing with primitive updates are:
*predConsq(T, u*p(inscrt*(pred,* T), 5), 5) —
prcdNegConsq(T, up(delete(prcd, T), 5), *S)* <—
Further rules define the effect of compound updates. If *pred* is a derived predicates, then the rules are extracted from the rules defining *pred.* A full definition of both predicates is in [Wal90].

The difference between *predSem* and *predConsq* can be illustrated using the example of the last section. The query "does r(a,b) hold after an update insert(p,X)?" yielded, the answer "yes, if $q(b) \wedge (X = a \vee p(a))$". The question "is r(a,b) an update consequence?", requires an

evaluation of the goal
$rConsq(a, b, up(insert(p, X), state1))$. Unfolding the rules for $rConsq$ we obtain the simpler answer "yes, if $q(b) \wedge X = a$".

4.4 How to Compile Integrity Checking

The compilation of integrity checking into update procedures introduces two requirements:

- the requirement to logically optimise the resulting integrity constraint check
- the requirement to express the resulting check as a standard knowledge base query against the original state

4.4.1 Partial Evaluation and Logical Optimisation

The first requirement, that of logical optimisation is impossible without unfolding, or partially evaluating, the goal. Partial evaluation of rulesets with negation is a relatively new technique. Our implementation is based on an algorithm developed by the author with David Chan and described in [CM89].

The result of partial evaluating a query *Q* is a set of qualified answers of the form $Q \leftarrow G_1, \ldots, G_n$. Each goal G, is either or literal, or a compound goal $\neg\exists X.(G_{i1}, \ldots, G_{iNi})$. Logical optimisation is the removal of inconsistent qualified answers, or redundant goals within an answer. If, for example, a qualified answer contains a subgoal *G* and its negation $\neg G$, then the answer is dropped. Similarly if it contains a goal *p{a)* and a compound goal $\neg\exists X.p(X)$ then again the answer is dropped. For theoretical and practical reasons, the optimiser cannot eliminate all inconsistencies in all cases, but in test cases our implementation lias proved to be effective.

For example suppose we have the following rules:
$s \leftarrow t(X), \neg v(X)$
$t(X) \leftarrow w(X, a)$
$v(X) \leftarrow w(X, Y)$
The result of partially evaluating *s* is
$s \leftarrow w(X, a) \wedge \neg\exists Y.w(X, Y)$
The system detects the inconsistency, and recogises that *s* must fail.

After logical optimisation, new predicates are introduced to replace the compound goals. Thereby the result of partial evaluation and logical optimisation is transformed back to the standard rule syntax. Specifically, for each compound goal $\exists \mathbf{X}$.*Goal,* a new predicate (say *np)* is introduced, with rule *np(Y) ← Goal,* where Y denotes the variables in *Goal* which do not appear in X. Now the compound goal is replaced by the atomic goal *nP(Y).*

4.4.2 Translating the Checks Back to Knowledge Base Queries

The final requirement for our compilation is the elimination of rules which "reason over multiple states". Since an update can involve only finitely many intermediate states, all the goals involving base predicates can be unfolded until the remaining calls are all against the original state. Now the "semantic" predicates can be unfolded using the rule RI described earlier. In case no recursion or negation occurs in the rules defining derived predicates, all goals can be unfolded, yielding a final qualified answer whose body contains only calls to base predicates, which can be unfolded as above.

In case recursion or negation does appear in the qualified answers, some new predicates have to be introduced, and rules added to the knowledge base defining them. However with these new rules, a condition can be derived which is also expressed on the original knowledge base state and which is necessary and sufficient to guarantee that no constraints will be violated.

For example if *anc* is recursively defined in terms of *p,*
$anc(XZ) \leftarrow p(X, Y), anc(Y, Z)$
$anc(X, Y) \leftarrow p(X, Y)$
and if an update procedure has the form
insert(p, *(X, Y))* then foreach A' : *anc(X,c)* do *U*
then the condition *anc(X,c)* must be invoked in the updated state up(insert(p, (X, *Y*)), *.state]*). The resulting semantic goal is
ancSem{X, c, up(insert(p, (A', V')), *.state])).*
The problem is to eliminate this goal in favour of one which can be evaluated in the original state *state].*

By a renaming procedure, similar to that used for eliminating compound goals, the system replaces the above goal by a goal and(X', Y, *A,c*), for which the following rules are automatically generated:
$\mathbf{ancl}(X, Y, X, C) \leftarrow \mathbf{ancl}(X, Y, Y, C)$
$ancl(X, Y, A, C) \leftarrow p(A, B), ancl(X, Y, B, C)$
$ancl(X, Y, X, Y) \leftarrow$
$ancl(X, Y, A, C) \leftarrow p(A, C)$
Having added the rules for *and* to the deductive database (at compile time), it is possible to evaluate the goal *and(X,* Y, *A, c)* in the original state.

Thus in all cases semantic goals are replaced in the compiled constraint check by standard queries expressed against the original knowledge base state.

5 Conclusion

In this paper we have studied the problem of optimised integrity checking for update procedures in deductive knowledge base systems. The method is based on an axiomatisation of the update language, expressed as semantic rules. Conditions for consistency of the new state with the integrity constraints after execution of an update procedure are expressed as a goal in the semantic language, which can be evaluated against these semantic rules.

We show how such a goal is partially evaluated and the resulting set of qualified answers is reduced by logical simplification. Finally the simplified set of qualified answers is translated back into the original knowledge base query language. The resulting set of conditions are imposed as preconditions on the update procedure. They are conditions on the original knowledge base which are necessary and sufficient to guarantee that any update using this procedure preserves integrity.

Of particular importance is the logical simplification, which corresponds to an optimisation step. For example if an update procedure is guaranteed to satisfy a certain constraint, whatever arguments are supplied at update time, then the precondition performs no checking related to the constraint. Effectively the consistency is detected at compile time, and the check removed during logical simplification.

A benefit of the compilation of constraints into update procedures is that the behaviour of the system when integrity is violated can be parameterised on the procedure which caused the violation.

The techniques employed here have shown how logical reasoning can be used effectively, even for procedural update languages like the one introduced in this paper. The approach was designed, and first applied, for the compilation of constraints into "safe" methods in an object knowledge base [Wal89]. However it has become clear that its applicability is not restricted to any particular data model.

6 Acknowledgements

The paper emerged from discussions about the way integrity constraints were implemented in the KB2 knowledge base system, by the author together with Hendrik Decker. Therefore Hendrik has had a fundamental influence on this work. Rodney Topor inspired me to start putting my thoughts on paper, and criticisms from Rainer Manthey and Francois Bry over the last year have ironed out some (though surely not all) of the wrinkles! My thanks to the above and to 1CL, Bull and Siemens for their funding of ECRC.

References

[Abi88] S. Abiteboul. Updates, a new frontier. In M. Gyssens, J. Paradaens, and D. Van Gucht, editors, *Proc. 2nd Int. Conf on Database Theory,* Lecture Notes in Computer Science 326, pages 1-18, Bruges, 1988. Springer-Verlag.

[BDM88] F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In *Proc. EDBT'88,* Venice, Italy, March 1988.

[BMM90] F. Bry, R. Manthey, and B. Martens. Integrity verification in knowledge bases. Technical Report D.2.1.a, ECRC, 1990.

[CM89] D. Chan and Wallace M.G. A treatment of negation during partial evaluation. In Abramson and Rogers, editors, *Meta Programming in Logic Programming,* 1989, MIT Press.

[Dec86] II. Decker. Integrity enforcement on deductive databases. In L. Kerschberg, editor, *Proc First International Conf On Expert Database Systems,* pages 271-286, Charleston, South Carolina, April 1986.

[Kue90] V. Kuechenhoff. On the efficient computation of the difference between consecutive database states. Tech. Rep. IR-KB-VK, ECRC, 1990.

[LST87] J. W. Lloyd, E. A. Sonenberg, and R. W. Topor. Integrity constraint checking in stratified databases. *JLP,* 4(4):331-343, 1987.

[Nic82] J -M. Nicolas. Logic for improving integrity checking in relational databases. *Acta Informatica,* 18:227-253, 1982.

[RBS87] R. Ramakrishnan, F. Bancilhon, and A. Silberschatz. Safety of recursive horn clauses with infinite relations. In *Proc. 6th PODS,* 1987.

[SK88] F. Sadri and R. A. Kowalski. A theorem-proving approach to database integrity. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming,* pages 313-362. Morgan Kaufmann, 1988.

[Sto75] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *Proc. ACM SIGMOD Conf on Mgi. of Data,* May 1975.

[Wal89] M.G. Wallace. Objects and integrity constraints. Tech. Rep. 1R-KB-70, ECRC,'1989.

[Wal90] M.G. Wallace. Compiling integrity checking into update procedures. Tech. Rep. IR-KB-81, ECRC, 1990.