Integrity constraints and constraint logic programming[*]

Henning Christiansen
Roskilde University, Computer Science Dept.,
P.O.Box 260, DK-4000 Roskilde, Denmark
henning@ruc.dk

Abstract

It is shown that constraint logic is useful for evaluation of integrity constraints in deductive databases. Integrity constraints are represented as calls to a meta-interpreter for negation-as-failure implemented as a constraint solver. This procedure, called lazy negation-as-failure, yields an incremental evaluation: It starts checking the existing database and each time an update request occurs, simplified constraints are produced for checking the particular update and new constraints corresponding to specialized integrity constraints are generated for the updated database.

1 Introduction

There is a relationship between integrity constraints in databases and the constraints of constraint logic programming going beyond the partial overlap of the names applied for these phenomena. Both concern conditions that should be ensured for systems of interdependent entities: the different tuples in a database, and the set of variables in a program execution state. Both relate to problems that evolve gradually: integrity must be ensured each time a database is updated, and the constraints of logic programming wake up to ensure satisfiability whenever more specific information becomes available about their arguments, by further instantiation or new constraints added to the constraint store.

Constraint logic programming is a computational paradigm which extends logic programming with incremental properties and makes it possible to cope with some of the problems of "classical" Prolog programming of over-generation and indefinite backtracking — without much compromise of a declarative style of programming.

It seems obvious to transfer these properties to the domain of integrity constraints in databases that always has been a source of difficulties in implementation and inefficiency. Integrity constraints, seen as global statements to be satisfied by a database, are avoided in most available database systems and applications although often partly compensated by hand-coded update routines.

[*]Invited talk at DDLP'99; appears in INAP'99 proceedings.

In the present paper, we make an experiment of using constraint logic for evaluation of integrity constraints. We make a restriction to deductive databases with only positive literals (however, allowing nonequalities in their bodies) and only positive updates. We apply a metaprogramming setting in which metavariables can stand for prospective updates and suggest a meta-level implementation of negation-as-failure adapted by means of constraints for evaluation of integrity constraints. The basic idea is straightforward: The procedure evaluates as far as possible, and whenever it runs into a branch where the presence of a metavariable (representing a yet unknown update request or sequence of such) makes it impossible to proceed, it delays. We call this principle *lazy negation-as-failure*. It turns out to be well suited for the purpose and it provides an incremental evaluation mechanism that casts of specialized integrity constraints for each update similarly to what is obtained by earlier methods called simplification (references and comparison given in the final section of this paper). The experience reported in the present paper indicates that constraint logic is well-suited for evaluation of integrity constraints, combining conceptual simplicity with desirable computational properties — and this holds whether we consider constraint logic programming (and deductive databases in Prolog for that matter) as the real topic or as a researchers' playground for designing and experimenting with new methods perhaps applicable in other technologies.

In section 2 we describe the overall setting and show different applications made with an implementation of the lazy negation-as-failure procedure. We consider generation of specialized update routines for particular updates, view update, and generation of intentional answers.

An implementation made using the language of Constraint Handling Rules [10] is described in section 3, and possible improvements and more efficient implementation methods are discussed. The final section 4 provides a summary and compares with related work.

2 Basic concepts and motivating examples

We illustrate the overall principle by examples made with an experimental implementation and we postpone as many details as possible about its implementation for section 3.

Databases are represented in the object language of function-free positive Horn clauses extended with `=` and `dif` constraints. The semantics is defined in terms of a provability relation $\vdash$ for ground queries with `=` and `dif` representing syntactic equality and nonequality. Clauses of the form `bottom:-` $\cdots$ are called integrity constraints and a database DB is considered consistent whenever $DB \not\vdash$ `bottom`.

For checking this property, we apply a metainterpreter `fails(`P'`, `Q'`)`, where P' and Q' are names of object program P and query Q. Its intended meaning is that Q (finitely) fails in P, i.e., we do not have $P \vdash Q\sigma$ for any object substitution σ. The `fails` predicate is implemented by means of a constraint solver and the property which makes it interesting for our purposes, is that it produces specialized and optimized versions of the integrity constraints for possible metavariables in P standing for future updates.

We use a ground representation [14, 13] of programs and queries with each symbol named by itself, except for object variables that are named by constants `'A'`, `'B'`, ..., `'Z'`. The following Prolog term names an object program to be used in the examples.

DB_0 =
```
  [(father(john, mary):- true),                (f1)
   (mother(jane,mary):- true),                 (f2)
   (bottom:- dif('A','B'),                     (ic1)
       father('A','C'), father('B','C')),
   (bottom:- dif('A','B'),                     (ic2)
       mother('A','C'), mother('B','C')),
   (bottom:-                                   (ic3)
       father('A','Z'), mother('A','X'))]
```

This database consists of facts about `father` and `mother` relations and three integrity constraints telling that you can only have one father and only one mother, and that a father cannot be a mother. Notice that the two `father` atoms in (ic1) occur symmetrically in the following sense: The same pair of tuples t_1, t_2 can be generated in two different ways but whether or not t_1 and t_2 give rise to an inconsistency is independent of which tuple is generated by which atom. The same holds for the two `mother` atoms in (ic2).

Checking the integrity of DB_0 can be done by calling `fails(`DB_0`, bottom)` that succeeds without leaving any unresolved constraints. In a case like this, the underlying procedure works similar to negation-as-failure in Prolog, examining all possible branches of execution the goal `bottom`.

2.1 Obtaining specialized integrity constraints

The ground representation makes it possible to indicate a future update by means of uninstantiated metavariables (i.e., Prolog variables) as follows; the ampersand is an operator for concatenation of different object programs. In the following call, `Clause` is a metavariable which stands in the position of an arbitrary clause which can be added to the program at a later stage.

`fails(`DB_0` & [Clause], bottom)` (Q_0)

The procedure works as far as possible from the input given to it, which means that it will process (and accept) the existing database DB_0, but those branches that depend on `Clause` have to delay. The processing of each of (ic1–3) gives rise to delayed constraints ready for receiving possibly new `father` or `mother` tuples that might imply an inconsistency, and another delayed constraints is ready to test a possible new integrity constraints that might arrive as `Clause = (bottom:- ...)`.

A more interesting behaviour can be observed when more specific information about the particular update is given which makes it possible for the constraint solver to simplify the result considerably. This can be given as a partial instantiation combined with auxiliary constraints as in the following query.

`constant(A), constant(B),` (Q_1)
`fails(`DB_0` & [(father(A,B):- true)], bottom).`

The two `constant` constraints are used in order to specify that the update must be a `father` fact. Declaratively, the meaning of `constant(`t`)` is that t needs to be the name of an object constant (as opposed to a name of an object variable). Executing (Q_1), the procedure considers again all branches, verifies the known part of the database, but now the constraint solver effectively prunes those branches that appeared in the processing of (Q_0) for possible new `mother` and `bottom` clauses. The following resulting constraints are returned as answer, ready to evaluate a future update given by values of `A` and `B`.

```
fails1(jane=A),                                 (A1)
fails1((mary=B,dif(john,A)))
```

The meaning of a `fails1` constraint is that the query named by its argument must fail under any instantiation of the possible metavariables. The first one is a remains of (ic3) combined with (f2) stating that a no new father can bear the name `jane` since `jane` already is registered as a mother. The second comes from (ic1) and (f1) and the conjunction inside `fails1` means that one of the two conditions must fail. Thus this constraints can be read: If the child mentioned in the update is `mary`, then the father must be `john`. In this way, the constraint solver has produced a specialized procedure expressed by (A_1) for verifying the update reduced into a form that do not refer to the database.

It is usually required that a database has no duplicate records. This can be expressed as a constraints which unrolls into a bunch of smaller constraints pairing each two clauses in the database. Most of them vanish immediately and for (Q_1), only `fails1((A=john,B=mary))` remains, expressing the condition for (f1) and the yet unknown fact. Taking this into account, the constraint solver can perform a further simplification step and reduce (A_1) into

```
fails1(B=mary), fails1(jane=A).                 (A'1)
```

In this particular example, the constraint solver has produced a specialized and optimal procedure for checking a single update of the `father` relation to DB_0. In section 3.3 we will discuss circumstances under which it is not desirable to unfold the constraints as far as done here.

2.2 Incremental integrity checking for sequences of updates

In the call

```
fails(DB0 & F, bottom)                          (Q2)
```

the metavariable `F` stands for a list of updates which can arrive one after another by a gradual instantiation of `F`. To begin with, we consider the case where all updates are known to be new `father` facts. In order to make it possible for the constraint solver to make use of this information we use the following constraint:

```
clause_pattern(F, (father(X,Y):- true),     (C)
                  (constant_(X), constant_(Y)))
```

The meaning is that any clause that may arise in `F` must match the given structure and satisfy the indicated constraints.[1] The constraint solver can use this constraint in order to prune branches that otherwise need to wait for an instantiation of `F`. For example, a `mother` atom fails with any clause that can arrive in `F`.

The execution of the query Q'_2:= (Q_2, C) will again process the existing database and leave specialized residual constraints for checking future updates. In this case, the constraints are slightly more complicated that those of (A_1), now formed by a three-argument version of `fails1`. The declarative meaning of

`fails1(`*Prog*`, ` *Atom*`*`*Cs*`,`*Continue*`)`

is that for each clause *H*`:-` *B* in *Cs*, the query *H*=*Atom*`,`*B*`,`*Continue* must fail in program *Prog*.

The following constraints result from the execution of Q'_2; variables with subscripts are generated by the system and renamed here so that they are easier to relate to the initial integrity constraints of DB_0. It will be explained in section 3 how the constraint solver has managed to replace object variable names by Prolog variables.

```
fails1(DB0 & F,father(A1,Z1)*F,mother(A1,X1)),
fails1(DB0 & F,father(B2,mary)*F,
                                   dif(john,B2)),
fails1(DB0 & F,
   (father(A3,C3),father(B3,C3))*F,dif(A3,B3))
```

The first one is a specialized version of (ic3) for `F`, the second a version of (ic1) for testing that no new clause in `F` produces a conflict with (f1). The third constraint in the answer displays a variant form used for literals that occur symmetrically in an integrity constraint, so that `(`*Atom*$_1$`,`*Atom*$_2$`)*`*Cs* indicates that *Atom*$_1$`,`*Atom*$_2$ are tested pairwise with all clauses of *Cs*, however, leaving out symmetric variants. Thus this constraint is a specialized version of (ic1) for testing that no inconsistency occurs within `F`.

Adding a tuple, say `father(john,peter)`, to the database is done by setting `F=[(father(john, peter):-true) | F1]` where `F1` represents future updates. This event starts the three constraints shown above which makes them cast off smaller constraints similar to those shown in section 2.1 for verifying `father(john,peter)`. The answer constraints mutate into the following, now also with a specialized version of (ic1) for testing that no new clause in `F1` conflicts with the newly added fact; we use *D* to abbreviate DB_0 `& [(father(A, B):-true) | F1]`.

```
fails1(D,father(A4,Z4)*F,mother(A4,X4)),
fails1(D,father(B5,mary)*F,dif(john,B5)),
fails1(D,father(B6,peter)*F,dif(john,B6)),
fails1(D,
   (father(A7,C7),father(B7,C7))*F,dif(A7,B7))
```

A scenery for updating with `mother` as well as with `father` tuples can be specified as follows.

```
clause_pattern(F, (father...), ...),            (Q3)
clause_pattern(M, (mother...), ...)),
fails1(DB0 & F & M, bottom)
```

The constraints produced follow the same pattern as already shown.

2.3 Integrity constraints in view updating

The procedure specified by the `fails` predicate can also be used together with abductive procedures for performing view update. In [3] we have described a metainterpreter defined by the predicate `demo(`P'`,` Q'`)`, which holds when P' and Q' are names of object program P and query Q with $P \vdash Q\sigma$ for some object substitution σ. Using the same sort of constraints as described in the present paper, this predicate can work with partly instantiated argument and serve as a program synthesizer. Let

```
DB1 = DB0 &
  [(sibling('X','Y'):- dif('X','Y')),        (f3)
      parent('Z','X'), parent('Z','Y'))
   (parent('X','Y'):- father('X','Y')),      (f4)
   (parent('X','Y'):- mother('X','Y'))].     (f5)
```

Thus, `sibling` is a view predicate in the sense that it is defined indirectly from the set of basic `father`

[1] In fact, the call to `clause_pattern` applies a nonground representation of the object language so it must be assumed that the variables `X` and `Y` in the example do not occur elsewhere in the query. Without this liberty, it had been necessary to develop an extra level of conventions for naming.

and `mother` facts in the database and no explicit `sibling` facts are allowed. The problem of updating the database in order to accumulate the fact `sibling(bob,mary)` can be stated by the query

```
clause_pattern(F, (father...), ...),          (Q4)
clause_pattern(M, (mother...), ...)),
fails(DB1 & F, bottom),
demo(DB1 & F, sibling(bob,mary)).
```

The underlying constraint solver provides an optimal interleaving of the computations performed by `demo` and `fails` and the following two alternative answers are produced,

`F = [(father(john,bob):-true) | F1]`, or
`M = [(mother(jane,bob):-true) | F1]`

each followed by a set of delayed constraint similar to those shown above for "direct" updates. In [4] it is described how this principle can be combined with simple induction problems in various ways. We can refer to [8] for another recent method which integrates integrity checking and view update.

2.4 Other applications

In a system intended to serve as a cooperative agent, it can be relevant to enter a dialogue with the user in order to find out why a particular database query or update request fails — and how it might be changed in order to succeed. See [20] for a recent overview of such systems.

The constraint-based `fails` predicate can also be used for these sort of things. Consider, for example, the query `mother(june, mary)` that fails when evaluated towards the database e.g., using `demo(`DB_0`, mother(june, mary))`. This failure arises since `mother(june, mary)` is not present in the database. In order to check whether this property, so to speak, is by accident or "by nature", we can check if an update with `mother(june, mary)` is possible:

```
fails(DB0 & [(mother(june, mary):- true)],
                                      bottom)
```

It fails, showing that `mother(june, mary)` violates the integrity constraints, i.e., it is not possible for `june` to become a mother of `mary`. In order to obtain more specific information we can try out different generalizations of the query in order to get more information, e.g.,

```
constant(A),
fails(DB0 & [(mother(A, mary):- true)],
                                      bottom).
```

Leaving out the check for no duplicate tuples, we get as single answer `A=jane`. So one way to explain why `june` cannot be a `mother` of `mary` is that `jane` is the only possible mother.[2] Alternatively, the other generalization

[2]Notice that an evaluation of `mother(A, mary)` against the database also gives answer `A=jane`, but this does not give evidence that `jane` is the *only* one possible.

```
constant(A),
fails(DB0 & [(mother(june, A):- true)],
                                      bottom)
```

yields `fails1(mary=A)` which tells that `june` can be assigned mother of any child, except `mary`.

It is also clear that the specialized constraints produced in this way can be used for semantic optimization of a query. If, for example, the task is to evaluate the query `mother(june,A)`, the example above shows that `fails1(mary=A)` can be added the query, perhaps limiting the search space. However, it still needs to be investigated for more interesting examples how much time is saved, taking into account the time spent on executing `fails`.

3 A constraint-based implementation of lazy negation-as-failure

The procedure used for evaluating calls to `fails` is a metainterpreter implemented by means of Constraint Handling Rules, CHR, which is an extension to Prolog making it possible to write constraint solver in a straightforward way; we refer to [10, 11] for background and introduction to CHR. We show these rules in an abstract way which can be rewritten straightforwardly into "real" CHR by means of auxiliary predicates for doing elementary data structure operations.

As mentioned, the constraint solver works with a ground representation of the object language for which the constraint `instance(`T_1`,`T_2`)` is extremely useful. Its meaning is that T_1,T_2 name object terms t_1,t_2 for which there exists an object substitution σ with $t_2 = t_1\sigma$, e.g., as in `instance(p('X'), p(a))`. These constraints embody a reflection of object variables into metavariables. For example, `instance(p('X'),Z)` is equivalent to `Z=p(Z1)` where `Z1` is a new metavariable. This means that the unification of two object terms can be simulated at the metalevel (i.e., Prolog) by two `instance` constraints followed by a metalevel unification. See [3] for a detailed analysis and description of a constraint solver, and [6] for a version in CHR.

At the top-level, the metainterpreter is defined as follows.

```
fails(P,Q) :- instance(Q,Q1), fails1(P,Q1).(0)
```

where the intended meaning of `fails1(P,Q1)` is that $P \not\vdash Q_1$, where P and Q_1 are the program and query named by `P` and `Q1`.

We show firstly an implementation of `fails1` which is intended for cases with completely specified arguments and then describe those adjustments that are necessary in order to cope with the general case.

3.1 A metainterpreter for known object programs and queries

The overall principle in the processing of calls to `fails1` is to select and process object literals one after another in its query argument (i.e., the second one). If the selected item can be recognized as failing, `fails1` succeeds immediately, otherwise we have to consider the rest of the query for the solutions that exist for the selected item.

If all items considered in the query has succeeded, failure is impossible:

```
fails1(_,true) <=> fail.                          (1)
```

For an equation, the test for satisfiability and the instantiation of possible solution is effectively performed by a metalevel unification:

```
fails1(P, (Q1,(A=B),Q2)) <=>                      (2)
  (A=B -> fails1(P, (Q1,Q2)) ; true)
```

The object level nonequality constraint is treated similarly to the `dif` predicate found in some Prolog dialect: If enough information is present to tell the arguments either identical or nonunifiable, the call is executed, otherwise it is delayed. In addition, if one (or both) of the arguments is a metavariable that does not occur in atoms or equations in the remaining query (we call such a variable *free* in the query), the nonequality is satisfiable and can be discarded from the query.

```
fails1(P, (Q1,dif(A,B),Q2)) <=>                   (3)
  A==B or both are constants or
    one is a variable free in (Q1,Q2) |
  (A==B -> true ; fails1(P, (Q1,Q2)))
```

For an atom to fail, it must fail with all clauses in the object program:

```
fails1([C1,...,Cn], (Q1,A,Q2)) <=>                (4)
  A names an object atom |
  fails1_with_clause(C1,P,A,(Q1,Q2)),
  ...
  fails1_with_clause(Cn,P,A,(Q1,Q2)).
```

In practice, we use the auxiliary constraint `fails1` with three arguments for the distribution over the clauses which, as shown in section 2.2, delays on uninstantiated program tails. In addition, it distributes over the ampersand operator and makes the mentioned optimization for symmetric literals.

Prolog's version of negation-as-failure uses backtracking as a way to avoid cluttering up the different unifications made to the variables in the selected atom and the different heads-of-clauses. In order to be able to delay subcomputations from the different branches of execution, we take a copy with new variables of the current object query.

```
fails1_with_clause((H:-B),P,A,Rest) <=>           (5)
  (predicates or arities of B and H different
   -> true
   ; copy_term((A,Rest), (A1,Rest1)),
     instance((H:-B),(H1,B1)),
     fails1(P,(H1=A1,B1,Rest1)).
```

where we use `H1=B1` as an abbreviation for the sequence of equations between pairs of arguments of `H1` and `B1`.

Local soundness and completeness of each of these rules is easy prove. The termination properties are similar to Prolog's negation-as-failure when we assume a computation rule selecting equations before anything else and atoms from left to right. The floundering problem in Prolog (or nonsound semantics in most versions) does not appear since any object variable in an object query given to `fails` is existentially quantified.

3.2 Coping with partly unknown object programs and queries

The most interesting applications of `fails` are when its arguments contain uninstantiated metavariables representing unknown parts of the object program or query. This means that some subcomputations need to be delayed while others can be reduced a bit further by the constraint solver from partial knowledge about these variables. We sketch here an adaptation of the procedure shown above; correctness statements still need to be formulated and proved.

As it appears in section 3.1, the terms naming object program and queries given to `fails` are "preprocessed" by `instance` constraints before `fails1` takes them into account in the computation process. This means that such metavariables, which we will call *external* variables, give rise to pending `instance` constraints and `fails1` works on other metavariables that will be affected if later, some event instantiates the external variable. An exception is made for external variables covered by a `constant` constraint which are copied directly into the terms processed by `fails1`. A variable is called *externally dependent* if it is external or it is covered by a pending `instance` constraint as indicated. Finally, we refine the notion applied in rule (3) above of a variable being *free* in a query argument only to concern variables that are not externally dependent.

The copy procedure applied in rule (5) needs to be made more precise: When a copy t' of a term t is made, any external variable in t should be preserved in t' and any `instance` constraints that can affect externally dependent variables in t should be copied in a similar way and added to the constraints store.

The presence of an externally dependent variable in an object equation or `dif` to be processed by `fails1` generally makes determination of a definite success or failure impossible. With a few exceptions, rules (2) and

(3) cannot apply in such cases. These are when the arguments are identical or one of them is free in the given query argument. Cases with opposite equations and `dif`'s occur quite often when integrity constraints are evaluated and are treated by a separate rule as follows (with additions to handle symmetric variants):

```
fails1(_,(...,X=Y,...,dif(X,Y),...)) <=> true.(6)
```

Whenever the second argument to `fails1` consists of equations and `dif`'s only, they become independent of the object program and we simplify them into a version of `fails1` with a single argument (cf. examples in section 2). These constraints are governed by rules similar to (2) and (3) plus some additional reduction rules including the following:

```
fails1((X=Y, C)), fails1((dif(X,Y), C)) <=>
    fails1(C).
```

A constraint of the form `fails1(dif1(`s`,`t`))` not removed by the rule similar to (3) is executed as a unification s=t and the rule (6) above can be extended also to take into account "global" constraints of the form `fails1(X=Y)`. Constraints `fails1((`C_1`,...,`C_n`))` that cannot be reduced further can be treated in two ways, either trying out each of `fails1(`C_1`)`, ... `fails1(`C_n`)` on backtracking or, which seems more convenient for the applications we consider in the present paper, leaving them in the state until further instantiation takes place. The treatment of one-argument `fails1` constraints gives our metainterpreter a flavour constructive negation [2].

The more complex constraints `fails1` (with two or three arguments) and `fails1_with_clause` delay whenever the argument standing for either the query or for the program (part) being traversed is uninstantiated. Here `clause_pattern` constraints are useful in order to have some of these calls succeed (and thus vanish from the constraint store) when the expected predicate symbols of the program part and atom under consideration are different.

3.3 Possible extensions and improvements

Due to the way `instance` constraints provide a translation of object variables into Prolog, it seems obvious to integrate our method with databases stored as Prolog facts. Instead of traversing the database as a list of represented clauses, they can be efficiently accessed by means of the indexing techniques usually applied in Prolog implementations. Here it will be of great advantage if we limit the unfolding of constraints as to avoid specializing, for example, (ic2) for the update pattern `mother(X,Y)` into one call `fails1((Y=`c`,dif(X,`m`)))` for *each* fact `mother(`m`, `c`)` in the database. Instead we should keep the call `fails1(...,mother(_,Y))` which, when the value of `Y` becomes known can be checked in more or less constant time. To achieve this, we can let (4) only apply for intentional predicates and delay any call to an extensional predicate that contains an externally dependent variable.

Similarly, we should avoid unrolling the condition for no duplicate records into quadratically many `dif1`'s and simply perform a database lookup as expressed by `fails1(...,`f`)` whenever a new fact f is suggested.

Another obvious optimization will be to compile the rules of the database into specialized versions of rule (5) and thus get rid of `instance` constraints and perhaps eliminate the program argument of `fails1` completely.

It seems also interesting to consider the approach adapted for commercially available database systems that can access very large amount of data in an efficient way. This can be imagined by means of an interface to Prolog or by building the functionality into the database system.

It seems also obvious that the metainterpreter can be speeded up substantially by implementing parts of it at lower levels such as modifying the underlying Prolog abstract machine. The repeated copying of query arguments performed by rule (5) is one example where operations directly at the internal data structures seem to be advantageous. The use of lower level message passing techniques for triggering the rules seems also to be a way to reduce the overhead introduced by CHR's loop searching for rules to apply.

4 Summary and discussion of related work

It has been proposed to apply constraint logic, in the shape of a constraint-based metainterpreter for lazy evaluation of negation-as-failure, for evaluation of integrity constraints in deductive databases. The procedure is lazy in the sense that it delays subcomputations on metavariables standing for potential updates, and this leads to an incremental execution: Instead of evaluating integrity constraints from scratch, only a minimal amount of work is made for each update.

The technique called simplification introduced by [19] for relational databases provides a way to specialize integrity constraints into simpler ones for specific updates under the assumption that the existing database satisfies these constraints. The specialized integrity constraints produced by our lazy negation-as-failure procedure appear quite similar although they are produced, so to speak, as a by-product of checking the entire database.

The simplification methods has been adapted for deductive databases by [18] (also described in [17]). This includes a simple bottom-up processing of the database rules starting from the updated predicates in order to identify an upper limit for the part of the database (including integrity constraints) that can be affected by the update, and thus needs to be processed in order to check that consistency has been preserved. The lazy negation-as-failure procedure's top-down processing of the database gives a similar (and actually more "precise") effect in its incremental processing of the evolving database: The branches of the evaluation of the integrity constraints that are not affected by the update have already been processed (or pruned) in the previous examination of the original database and partial processing of the update (e.g., from knowledge about its predicate). See also the analysis of [18]'s method made by [16]. Later extension and improvements of the simplification method are reported by [7, 21, 9, 8].

A method, which is related to ours is [16], applying partial evaluation of a metainterpreter in order to obtain specialized (procedures for checking) integrity constraints and a considerable improvement of efficiency compared with [18] is reported. The techniques of constraint solving and of partial evaluation are related in the sense that both produce specialized code waiting to process the remaining input, in the form of either delayed and reduced constraint or a residual program, typically with new specialized predicates. Constraint solving is conceptually simpler, being essentially a declarative programming paradigm, whereas partial evaluation permits a detailed control of, say, unfolding of predicate calls.

We see our main contribution as showing that constraint solving is a technique that is relevant for checking of integrity constraints since 1) it matches the declarative nature of integrity constraints, and 2) that it provides the necessary incremental evaluation in order to avoid checking the entire database each time an update arise. We see this and the partial evaluation based method of [16] as two complementary techniques that can make benefit of each other. As one example, it seems obvious to apply partial evaluation in order to eliminate the metainterpretation layer and compile database clauses and integrity constraints into specialized Prolog and CHR rules for doing the checking.

It is also clear from our work that constraint solvers for lazy negation-as-failure can be designed in many different ways and optimized for different classes of logic programs. An obvious shortcoming of the version described in the present paper is the lacking ability to handle nested negations. However, we see this more as a property of the present implementation than a problem in the use constraints. Inspiration for an improvement can be found an earlier metainterpreter suggested by [15] which handles nested negations by means of constructive negation[2] in a straightforward way, however, not in a context where metavariables can represent unknown parts of object programs and queries. It seems interesting to attempt a generalization of it by means of the techniques described in [3] and the present paper. Another way to represent negation in the object language is by means of so-called explicit negation with, for each predicate p, to assume another predicate not_p with an integrity constraint to preserve consistency.

In [5, 6] we describe a generalization of lazy negation-as-failure to programs with function symbols as it appears in DemoII system. That version, however, is more complicated and tends to delay more often since we cannot process object equations and nonequations with the same ease as in the present context.

Another and deeper problem concerns update by deletions. The representation of object programs as datastructures with changes given as instantiation of metavariables implies that the approach is monotonic. Information can only be added, not remove which means that deletions have to be represented symbolically so that, e.g., the deletion of `father(john,mary)` is given as the addition of a fact `deleted_father(john,mary)`.

Finally, we refer to the following overview papers on integrity constraints and their applications [1, 12].

Acknowledgments

This research is supported in part by the DART project funded by the Danish Research Councils. Thanks to Davide Martinenghi for invaluable discussions and experimentation with different implementations of the procedure.

References

[1] F.Bry, R.Manthey and B.Martens: Integrity verification in knowledge bases. In A.Voronkov (ed.): Proc. Second Russian Conference on Logic Programming. *Lecture Notes in Computer Science* 592, pp. 114-139, Springer (1992)

[2] D.Chan: Constructive negation based on the completed database. In R.A.Kowalski and K.A.Bowen (eds.): *Logic Programming, Proc., 5th Intl. Conference and Symposium*, pp. 111–125, MIT Press (1988)

[3] H.Christiansen: Automated reasoning with a constraint-based metainterpreter. *Journal of Logic programming*, vol. 37, pp. 213–253 (1998)

[4] H.Christiansen, Abduction and induction combined in a metalogic framework. In P Flach, A.Kakas (eds.): *Abductive and Inductive Reasoning: Essays on their Relation and Integration*, To appear, 1999.

[5] H.Christiansen and D.Martinenghi: *The DemoII system.* Source code for implemented system, example files, and manuals available by World Wide Web, http://www.dat.ruc.dk/software.htm (1998)

[6] H.Christiansen and D.Martinenghi, Symbolic constraints for meta-logic programming, *Journal of Applied Artificial Intelligence*, to appear (1999)

[7] H.Decker: Integrity enforcement on deductive databases, In L.Kershberg (ed.):*Expert database systems*, pp. 381–395, Benjamin-Cummings (1987)

[8] H.Decker: An extension of SLD by abduction and integrity maintenance for view updating in deductive databases. In: M.J.Maher (ed.): *Logic Programing, Proc. of the 1996 Joint Int'l Conference and Symposium on Logic Programming*, pp. 157–169, MIT Press (1996)

[9] H.Decker and M.Celma: A slick procedure for integrity checking in deductive databases. In P.v.Hentenryck (ed.): *Logic Programming: Proc. Eleventh Int'l Conference on Logic Programming*, pp. 456–469, MIT Press (1994)

[10] Frühwirth, T.W., Constraint handling rules, In: A.Podelski (ed.): Constraint Programming: Basics and Trends, Châtillon Spring School, *Lecture Notes in Computer Science* 910, pp. 90-107 (1995)

[11] Frühwirth, T.W., Theory and Practice of Constraint Handling Rules, *Journal of Logic Programming*, vol. 37, pp. 95-138 (1998)

[12] P.Godfrey, J.Grant, J.Gryz and J.Minker: Integrity Constraints: Semantics and Applications. In J.Chomicki and G.Saake (eds.): *Logics for Databases and Information Systems*, pp. 265–306 (1998)

[13] P.M.Hill and J.Gallagher: Meta-programming in logic programming. In D.M.Gabbay, C.J.Hogger, and J.A. Robinson (eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, vol. 5, pp. 421–498. Oxford University Press (1998)

[14] P.M.Hill and J.W.Lloyd: Analysis of meta-programs. In H.Abramson and M.H.Rogers (eds.): *Meta-programming in Logic Programming.* MIT Press, pp. 23–51 (1989)

[15] M.Johnson: A negation meta interpreter using antisubsumption constraints. Posted to comp.lang.prolog (1992)

[16] M.Leuschel and D.De Schreye: Creating specialised integrity checks through partial evaluation of meta-interpreters. *Journal of Logic programming*, vol. 36, pp. 149–193 (1998)

[17] J.W.Lloyd: *Foundations of Logic Programming, 2nd Edition*, Springer-Verlag 1987.

[18] J.W.Lloyd, E.A.Sonenberg, and R.W.Topor: Integrity constraint checking in stratified databases. *Journal of Logic programming*, vol. 4, pp. 331–343 (1987)

[19] J.-M.Nicolas: Logic for improving integrity checking in relational data bases. *Acta Informatica* 18, pp. 227–253 (1982)

[20] J.Minker: An Overview of Cooperative Answering in Databases. In T.Andreasen, H.Christiansen, and H.L.Larsen (eds.): Proc. Intl. Conference on Flexible Query Answering Systems, FQAS'98, *Lecture Notes in Computer Science*, pp. 282–285 (1998)

[21] F.Sadri and R.A.Kowalski: A theorem-proving approach to database integrity, In J.Minker (ed.): *Foundations of Deductive Databases and logic programming*, pp. 313–362, Morgan-Kauffman (1988)