

Detecting Unsolvable Queries for definite Logic Programs

Maurice Bruynooghe

Henk Vandecasteele

D. Andre de Waal

Marc Denecker

Report CW 270, July 1998



Katholieke Universiteit Leuven

Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Detecting Unsolvable Queries for definite Logic Programs*

Maurice Bruynooghe

Henk Vandecasteele

D. Andre de Waal

Marc Denecker

Report CW 270, July 1998

Department of Computer Science, K.U.Leuven

Abstract

In logic programming, almost no work has been done so far on proving that certain queries cannot succeed. Work in this direction could be useful for queries which seem to be non-terminating. Such queries are not exceptional, e.g. in planning problems. The paper develops some methods, based on abduction, goal-directedness, tabulation, and constraint techniques, for proving failure of queries for definite logic programs. It also reports some experiments with various tools.

*A version of this report will appear in the Proceedings of PLILP'98

Detecting Unsolvable Queries for Definite Logic Programs

Maurice Bruynooghe¹ and Henk Vandecasteele¹ and D. Andre de Waal² and
Marc Denecker¹

¹ Departement Computerwetenschappen, Celestijnenlaan 200A, Katholieke
Universiteit Leuven, B-3001 Heverlee, Belgium
e-mail: {maurice, henkv, marcd}@cs.kuleuven.ac.be,

² Centre for Business Mathematics and Informatics, Potchefstroom University for
Christian Higher Education, South Africa
e-mail: BWIDADW@puknet.puk.ac.za

Abstract. In logic programming, almost no work has been done so far on proving that certain queries cannot succeed. Work in this direction could be useful for queries which seem to be non-terminating. Such queries are not exceptional, e.g. in planning problems. The paper develops some methods, based on abduction, goal-directedness, tabulation, and constraint techniques, for proving failure of queries for definite logic programs. It also reports some experiments with various tools.

1 Introduction

Given some logic program, executing a query may have many different outcomes. It may terminate or run forever (in practice until some resource is exhausted). In both cases, the query may or may not lead to answers. There is a large body of literature on termination analysis (see [7] for a survey). However, termination conditions are not decidable and automated methods are based on analysing the size of syntactical structures. So there is a substantial class of programs for which automated termination analysis fails. For some of these, loop checking methods [2, 3, 15] monitoring the execution, bring some relief, by pruning some infinitely failing branches. Methods have to choose between pruning too much, causing incompleteness in the search for solutions, and preserving completeness but still allowing some infinite computations. Also an execution mechanism augmented with tabling such as XSB [14] or an approach such as [4] reduces the number of non-terminating queries.

So far, there are almost no works on program analysis which attempt to recognise (infinitely) failing queries. Problems with planners, which generate more and more complex objects until one is found with a particular property have been the incentive to start our research. It would be very useful to be able to stop their infinite search for a solution to an unsolvable problem. Also certain program properties can be proven by proving failure of a particular query. As a trivial example, consider a program which knows about even and odd numbers.

One can prove the program does not define a number which is both even and odd by proving that a query asking for such a number fails.

Conceptually, there is a simple way to show that a query must fail: find a model of the program in which the query is false. In [8] some of the authors of the current paper made a first exploration of the issues involved in finding such a model for definite programs. The current paper develops two methods for automating the search for a pre-interpretation underlying such a model. A first approach combines abduction and tabulation to direct the search for a pre-interpretation. A second approach considers the abducibles as constraints and uses techniques from finite domain constraint solving [20] to further prune the search. Also the suitability of alternative methods for solving this problem is analysed. The use of a general purpose model generation tool [16, 17] is evaluated. We have also explored whether tools for type inference [11, 5] can show that such queries have an empty success set and whether conjunctive partial deduction [13] can specialise such queries into a trivially failing program.

Section 2 recalls the basics about pre-interpretations and introduces a trivial example. Section 3, explains how a pre-interpretation can be described by a number of facts, how a program can be abstracted as a DATALOG program, and how the least model based on that pre-interpretation can be queried by evaluating the abstracted query on the DATALOG program. In section 4, a procedure combining abduction with tabulation and a variant handling the abducibles as constraints are developed. Other approaches we are aware of which can directly or indirectly prove failure are discussed in Section 5. In section 6, the different approaches are compared. Finally, in section 7, we draw some conclusions. We assume some familiarity with the basics of tabulation, e.g. [19, 14, 21].

2 Preliminaries

A pre-interpretation J of a program P consists of a domain $D = \langle d_1, \dots, d_m \rangle$ and, for every functor f/n a mapping f_J from D^n to D . An interpretation I based on a pre-interpretation J consists of a mapping p_I from D^n to $\{true, false\}$ for every predicate p/n in P . An interpretation is often identified by the set of atoms $p(d_1, \dots, d_n)$ for which $p_I(d_1, \dots, d_n)$ is mapped to *true*. An interpretation M is a model of a program P iff all clauses of P are true under the interpretation M . A definite program always has a model (map $p_I(d_1, \dots, d_n)$ to *true* for all predicates and all domain elements). The intersection of two models is also a model and there is always a unique least model. As a consequence, if an existentially quantified conjunction $\exists_{\bar{X}} L_1 \wedge \dots \wedge L_n$ is false in a model based on a pre-interpretation J then it is also false in the least model based on that pre-interpretation. So, given a pre-interpretation, it suffices for our purposes to evaluate the conjunction in the least model.

Example 1. Even/odd

`even(0). even(s(X)) ← odd(X). odd(s(X)) ← even(X).`

$D = \{\mathcal{E}, \mathcal{O}\}$

$0_J = \mathcal{E} \quad s_J(\mathcal{E}) = \mathcal{O} \quad s_J(\mathcal{O}) = \mathcal{E}$

The least model is $\{even(\mathcal{E}), odd(\mathcal{O})\}$. The query $\leftarrow \mathbf{even(X)}, \mathbf{odd(X)}$ fails because $\exists_X even(X), odd(X)$ is false in this model. Executing the program with SLD or with a tabulating procedure (e.g. XSB [14]) results in infinite failure.

3 Proof procedures

Pre-interpretations based on finite domains have been used in the analysis of logic programs. The approach was pioneered by Codish and Demoen [6] for groundness analysis and subsequently by others, e.g. by [10] for type analysis. They used *abstract compilation* and represented the functions f_J/n of the pre-interpretation by $n + 1$ -ary relations over the domain and replaced all terms in the program by their pre-interpretation. This gives a so called abstract program which is a DATALOG program. Its finite model expresses declarative properties.

Example 2. In Example 1, we can define the pre-interpretation by:

$0_J(\mathcal{E}). \quad s_J(\mathcal{E}, \mathcal{O}). \quad s_J(\mathcal{O}, \mathcal{E}).$

To abstract the program, non-variable terms are replaced by fresh variables which are defined by the appropriate relations (a term $f(t_1, \dots, t_n)$ is replaced by a fresh variable X and the atom $f_J(t_1, \dots, t_n, X)$ is added to the body; this construction is repeated until all terms have disappeared). Variables are left as they are, the effect of the abstraction is that they now range over the domain of the pre-interpretation. This gives the following program:

$\mathbf{even(X)} \leftarrow 0_J(\mathbf{X})$
 $\mathbf{even(Y)} \leftarrow s_J(\mathbf{X}, \mathbf{Y}), \mathbf{odd(X)}.$
 $\mathbf{odd(Y)} \leftarrow s_J(\mathbf{X}, \mathbf{Y}), \mathbf{even(X)}.$

The clauses together with the facts of the pre-interpretation are a DATALOG program. The least model is $\{0_J(\mathcal{E}), s_J(\mathcal{E}, \mathcal{O}), s_J(\mathcal{O}, \mathcal{E}), even(\mathcal{E}), odd(\mathcal{O})\}$. The formula $\exists_X even(X), odd(X)$ is false in this model. While the query $\leftarrow \mathbf{even(X)}, \mathbf{odd(X)}$ is nonterminating under SLD, it fails finitely under well known proof procedures such as bottom-up evaluation after magic-set transformation or top-down methods enriched with tabulation such as OLDT [19] and XSB [14].

4 The search for the right pre-interpretation

To prove that a query for a program P, which seems to run forever without returning a solution, fails, one has to select a domain and a pre-interpretation and has to show finite failure when executing the abstracted query with the abstracted program. A straightforward way consists of selecting a domain, and trying all pre-interpretations until one is found for which the query fails. If none exists, one can try again with a larger domain. However, for programs with a substantial number of function symbols and constants, this quickly results in a very large search space. Indeed, with a n -element domain, an m -ary functor has $n^{(n^m)}$ possible pre-interpretations.

4.1 An abductive approach

To prune the search one can employ an abductive procedure: execute the abstract program with an initially unknown pre-interpretation, and abduce the different components of the pre-interpretation when they are needed during the execution. As soon as the query succeeds, backtracking can be initiated. To do so, one declares the predicates $f_J/n + 1$ as abducibles, add constraints that the pre-interpretation of each functor f/n is a total function, i.e. that one fact $f_J(d_1, \dots, d_n, d)$ must be abduced for every combination $\langle d_1, \dots, d_n \rangle$ of domain elements and employ a general purpose abductive procedure such as SLDNFA [9]. Some initial experiments showed the feasibility¹, but also the need for a dedicated procedure which allows to experiment with different control strategies. We designed and implemented an abductive procedure for definite programs which makes use of tabulation and which has the integrity constraints on the pre-interpretation hardwired in the code. By doing some experiments we gained a better understanding of the issues which are important to control the search. For example we observed that it performed better when tabling only the most general call for each predicate. Below we describe the procedure as (inference) rules which map sets of clauses (a “state”) to sets of clauses. The control—in which order the different rules are applied—is left undetermined. The system attempts to abduce a pre-interpretation such that the query fails under a top-down execution with tabulation of the abstract program, i.e. is false in the corresponding model of the abstract program.

We need some notational conventions. The state of the computation is represented as a set of clauses. The symbol **C1** is used to represent a clause and the symbol **C1s** to represent a set of clauses. We use $\mathbf{C1} :: \mathbf{C1s}$ to represent the set of clauses $\{\mathbf{C1}\} \cup \mathbf{C1s}$. **A** and **B** are used to represent atoms, **As** and **Bs** to represent sequences of atoms. A clause is represented as $\mathbf{H} \leftarrow \mathbf{As}$ in which the head **H** is an atom (or **false**). Given a query $\leftarrow \mathbf{As}$, the initial state of the derivation is represented as $(\mathbf{false} \leftarrow \mathbf{As})$. \mathbf{p}/n refers to a predicate of the original program, calls to such predicates are tabled. $\mathbf{abduce}_f(\mathbf{t}_1, \dots, \mathbf{t}_n, \mathbf{X})$ is the notation we use for a call to an abducible predicate $\mathbf{f}_J(\mathbf{t}_1, \dots, \mathbf{t}_n, \mathbf{X})$ of the pre-interpretation. These calls are not tabled. In a state of the derivation (a set of clauses), the calls $\mathbf{p}(\overline{\mathbf{X}})$ which are tabled are represented implicitly through the occurrences of literals $\mathbf{Lookup}(\mathbf{p}(\overline{\mathbf{I}}))$ in the bodies of the clauses. The answers to tabled calls are represented by clauses with an empty body.

We assume a fixed number of domain elements. Rule 1 handles a new call to a program predicate. The call is wrapped inside **Lookup** to indicate that it is waiting for answers. Nothing else needs to be done when the predicate was called before. Otherwise, the clauses defining the predicate are added. Eventually, they will lead to facts which are answers to the most general call of the predicate. Rule 2 describes the *lookup* step: a (wrapped) call is unified with an answer and the resolvent is added (for simplicity of presentation, we assume the state remains the same when—up to renaming—the same clause is derived a second

¹ Because the procedure lacks tabulation, some extra transformation of the clauses of recursive predicates was required.

time). Rule 3 resolves an abducible with an already abduced fact and adds the resolvent. Rule 4 abduces a new fact. Several domain elements are available, so a *choice point* is created. Rule 5 detects that the query has an answer, i.e. that the chosen pre-interpretation does not satisfy, and triggers backtracking.

1. $H \leftarrow p(\bar{t}), \text{As} :: \text{Cls}$
 Let $p(\bar{X}_i) \leftarrow \text{Bs}_i$ ($i: 1, \dots, m$) be the clauses in the definition of p . If this is the first call to p (Cls does not contain a clause with an atom $\text{Lookup}(p(\bar{r}))$) then the new state is:
 $p(\bar{X}_1) \leftarrow \text{Bs}_1 :: \dots :: p(\bar{X}_m) \leftarrow \text{Bs}_m :: H \leftarrow \text{Lookup}(p(\bar{t})), \text{As} :: \text{Cls}$
 Else the new state is:
 $H \leftarrow \text{Lookup}(p(\bar{t})), \text{As} :: \text{Cls}$
2. $H \leftarrow \text{Lookup}(p(\bar{t})), \text{As} :: p(\bar{s}) \leftarrow :: \text{Cls}$
 where $p(\bar{s}) \leftarrow$ is a fact which unifies with $p(\bar{t})$.
 The new state is:
 $(H \leftarrow \text{As})\text{mgu}(\bar{t}, \bar{s}) :: H \leftarrow \text{Lookup}(p(\bar{t})), \text{As} :: p(\bar{s}) \leftarrow :: \text{Cls}$
3. $H \leftarrow \text{abduce}_f(\bar{t}), \text{As} :: \text{abduce}_f(\bar{s}) \leftarrow :: \text{Cls}$
 where $\text{abduce}_f(\bar{s}) \leftarrow$ is a fact unifying with $\text{abduce}_f(\bar{t})$.
 The new state is (use of abduced fact):
 $(H \leftarrow \text{As})\text{mgu}(\bar{t}, \bar{s}) :: H \leftarrow \text{abduce}_f(\bar{t}), \text{As} :: \text{abduce}_f(\bar{s}) \leftarrow :: \text{Cls}$
4. $H \leftarrow \text{abduce}_f(\mathbf{t}_1, \dots, \mathbf{t}_m, \mathbf{t}), \text{As} :: \text{Cls}$
 Let $\mathbf{d}_1, \dots, \mathbf{d}_m$ be domain elements which unify respectively with $\mathbf{t}_1, \dots, \mathbf{t}_m$ and such that Cls has not yet a fact $\text{abduce}_f(\mathbf{d}_1, \dots, \mathbf{d}_m, \mathbf{d}) \leftarrow$ for some domain element \mathbf{d} . A **choice point** is created. A domain element \mathbf{d} is selected and the new state is (abduction of a new fact):
 $\text{abduce}_f(\mathbf{d}_1, \dots, \mathbf{d}_m, \mathbf{d}) \leftarrow :: H \leftarrow \text{abduce}_f(\mathbf{t}_1, \dots, \mathbf{t}_m, \mathbf{t}), \text{As} :: \text{Cls}$
5. $(\text{false} \leftarrow) :: \text{Cls}$
Backtrack to the state corresponding to the most recent **choice point** with an untried domain element \mathbf{d} .

We have a proof that the query fails when the system reaches a stable final state (no new clauses can be inferred —up to renaming— and $(\text{false} \leftarrow)$ is not part of the state). The search for a proof fails when the rewriting fails (rule 4 has exhausted all choices for a tuple $\mathbf{d}_1, \dots, \mathbf{d}_m$ of domain elements). In the latter case, one could increase the size of the domain and start over. The application of the rules is nondeterministic. An obvious control strategy delays the introduction of choice points as long as possible². We have experimented with various search rules (selection of clause) and computation rules (selection of literal). In section 6 we give more details on the system which behaved best. Experiments revealed that none of the strategies is superior and that overall performance on a particular problem is rather dependent on the order in which the pre-interpretations of the different functors are abduced. This motivated the search for a better approach.

² Symmetries: with n domain elements, each pre-interpretation can be mapped into an equivalent one (for what concerns the truth of the query in the least model) by permuting the domain elements. Our implementations avoid most symmetries.

4.2 A constraint approach

From now on, we use a slightly different notation for the abstract program. The abstraction of an atom $p(\mathbf{f}(\mathbf{a}))$ is denoted as $\mathbf{abduce}(\mathbf{f}(\mathbf{a}), \mathbf{X}), p(\mathbf{X})$ (We use $\mathbf{abduce}(\mathbf{f}(\mathbf{a}), \mathbf{X})$ as an abbreviation of $\mathbf{abduce}_a(Y), \mathbf{abduce}_f(Y, X)$).

The problem with the abductive approach can be illustrated with the following example: Assume that the pre-interpretation of a functor $\mathbf{f}/1$ has already been abduced as $\mathbf{abduce}(\mathbf{f}(\mathbf{d1}), \mathbf{d1})$ and $\mathbf{abduce}(\mathbf{f}(\mathbf{d2}), \mathbf{d2})$ and that a clause $\mathbf{false} \leftarrow \mathbf{abduce}(\mathbf{f}(\mathbf{g}(\mathbf{h}(\mathbf{a}))), \mathbf{X}), \mathbf{abduce}(\mathbf{g}(\mathbf{h}(\mathbf{a}))), \mathbf{X}$ is derived. Whatever is the pre-interpretation for \mathbf{a} , \mathbf{h} , and \mathbf{g} , $\mathbf{false} \leftarrow$ will be derived. The abductive systems will not revise the pre-interpretation for \mathbf{f} before having *thrashed* over most of the pre-interpretations of \mathbf{a} , \mathbf{h} , and \mathbf{g} .

A constraint based approach can to a large extent avoid such problems. We consider the abducibles as constraints and use a special purpose constraint solver which checks the existence of a pre-interpretation which satisfies all constraints. In the above example, if the pre-interpretation of $\mathbf{f}/1$ is constrained to the shown one and the clause $\mathbf{false} \leftarrow \mathbf{abduce}(\mathbf{f}(\mathbf{g}(\mathbf{h}(\mathbf{a}))), \mathbf{X}), \mathbf{abduce}(\mathbf{g}(\mathbf{h}(\mathbf{a}))), \mathbf{X}$ is derived, then the solver detects the inconsistency and triggers backtracking.

This approach makes it necessary to reformulate our abductive system. The major difference is wrt. the tabulation. The answers to a tabled predicate are no longer ground facts but constrained facts (of the form $p(\overline{X}) \leftarrow \mathbf{abduce}(\dots), \dots, \mathbf{abduce}(\dots)$). A problem is that one can have an infinite number of syntactically different answers. However, with a finite domain and a fixed pre-interpretation, the set of answers (its model) is finite. So it must be possible to add constraints which enforce the finiteness. Before presenting the formal system, we illustrate the main ideas with the **even/odd** example.

Example 3. Even/odd

The program is as follows:

```

even(X) ← abduce(0, X).
even(Y) ← abduce(s(X), Y), odd(X).
odd(Y) ← abduce(s(X), Y), even(X).

```

We represent the state of the derivation by three components, the set of clauses, the set of answers and the constraint store, holding the set of constraints (as before the component with the fixed abstract program is left out). ϵ stands for the empty set. *Lookup* is abbreviated as L , and *false*, *abduce*, *even* and *odd* respectively as f , ab , e and o . Finally, *sb* is the abbreviation of *subsumed*. In the initial state ((0) in Table 1) the only clause is the query. The leftmost atom of a program predicate is selected and the two clauses defining **even/1** are activated (1). The second clause is a constrained fact. In principle, we have to create a choice point. The first alternative adds the constraint **subsumed**(**even**(X) ← **abduce**(0, X), { }) to the constraint store in an attempt to have the new fact subsumed by the existing ones. The constraint is false for every pre-interpretation, so the second alternative is taken: The fact is added to the set of answers and the constraint **not**(**subsumed**(**even**(X) ← **abduce**(0, X), { }))) is added to the store. The constraint is redundant wrt. the (empty) store, so

	clauses	answers	constraint store
0	$f \leftarrow e(X), o(X)$	ϵ	ϵ
1	$f \leftarrow L(e(X)), o(X)$ $e(X) \leftarrow ab(0, X)$ $e(Y) \leftarrow ab(s(X), Y), o(X)$	ϵ	ϵ
2	$f \leftarrow L(e(X)), o(X)$ $e(Y) \leftarrow ab(s(X), Y), o(X)$	$e(X) \leftarrow ab(0, X)$	ϵ
4	$f \leftarrow L(e(X)), o(X)$ $e(Y) \leftarrow ab(s(X), Y), L(o(X))$ $o(Y) \leftarrow ab(s(X), Y), L(e(X))$	$e(X) \leftarrow ab(0, X)$	ϵ
6	$f \leftarrow L(e(X)), o(X)$ $f \leftarrow ab(0, X), o(X)$ $e(Y) \leftarrow ab(s(X), Y), L(o(X))$ $o(Y) \leftarrow ab(s(X), Y), L(e(X))$ $o(Y) \leftarrow ab(s(0), Y)$	$e(X) \leftarrow ab(0, X)$	ϵ
8	$f \leftarrow L(e(X)), o(X)$ $f \leftarrow ab(0, X), L(o(X))$ $e(Y) \leftarrow ab(s(X), Y), L(o(X))$ $o(Y) \leftarrow ab(s(X), Y), L(e(X))$	$e(X) \leftarrow ab(0, X)$ $o(Y) \leftarrow ab(s(0), Y)$	ϵ
10	$f \leftarrow L(e(X)), o(X)$ $f \leftarrow ab(0, X), L(o(X))$ $f \leftarrow ab(0, X), ab(s(0), X)$ $e(Y) \leftarrow ab(s(X), Y), L(o(X))$ $e(Y) \leftarrow ab(s(s(0)), Y)$ $o(Y) \leftarrow ab(s(X), Y), L(e(X))$	$e(X) \leftarrow ab(0, X)$ $o(Y) \leftarrow ab(s(0), Y)$	ϵ
11	$f \leftarrow L(e(X)), o(X)$ $f \leftarrow ab(0, X), L(o(X))$ $e(Y) \leftarrow ab(s(X), Y), L(o(X))$ $e(Y) \leftarrow ab(s(s(0))), Y,$ $o(Y) \leftarrow ab(s(X), Y), L(e(X))$	$e(X) \leftarrow ab(0, X)$ $o(Y) \leftarrow ab(s(0), Y),$	$f \leftarrow ab(0, X), ab(s(0), X)$
12	$f \leftarrow L(e(X)), o(X)$ $f \leftarrow ab(0, X), L(o(X))$ $e(Y) \leftarrow ab(s(X), Y), L(o(X))$ $o(Y) \leftarrow ab(s(X), Y), L(e(X))$	$e(X) \leftarrow ab(0, X)$ $o(Y) \leftarrow ab(s(0), Y)$	$f \leftarrow ab(0, X), ab(s(0), X)$ $sb(e(Y)) \leftarrow ab(s(s(0)), Y),$ $\{e(X) \leftarrow ab(0, X)\}$

Table 1. Constraint based execution of even/odd

the store remains empty (2). The call **odd(X)** is selected in the second clause, the clause defining **odd/1** is added, and its atom **even(X)** is selected (4). The stored answer is used to resolve with the first and third clause, this results in two new clauses³ (6). In the first, the atom **odd(X)** is selected, the last is an answer for **odd/1**. We have a choice point but again the first alternative creates an inconsistent store and the second alternative a redundant constraint, so the net effect is that the fact is added to the answer set (8). This answer is used to resolve with the second and third clause, resulting in two new clauses (10). The first one is a constraint which is consistent with the current store and added to it (11),

³ Remark that $abduce(s(X), Y)$, $abduce(0, X)$ is abbreviated by $abduce(s(0), Y)$.

It means that 0 and $s(0)$ should be different under the pre-interpretation. The second is an answer for **even/1**. This time we have a real choice point. The first alternative enforces the constraint that the new answer is subsumed by the existing answers, so the answer is not stored and the constraint $\text{subsumed}(\text{even}(Y)) \leftarrow \text{abduce}(s(s(0)), Y), \{\text{even}(X) \leftarrow \text{abduce}(0, X)\}$, which is consistent with the current store is added to it. It actually means that 0 and $s(s(0))$ must be equal under the pre-interpretation. A stable state is reached so the query is false in the models of the pre-interpretations satisfying the constraints of the store. The pre-interpretation $\text{abduce}(0, d1), \text{abduce}(s(d1), d2)$ and $\text{abduce}(s(d2), d1)$ is a solution (the subsumption test reduces to $\text{subsumed}(\text{even}(d1), \{\text{even}(d1)\})$ and yields true). Observe that a $\text{not}(\text{subsumed}(\dots))$ constraint is added to the constraint store each time that a new answer is added to the answer set. The conjunction of these constraints enforces finiteness of the answer set and termination of the algorithm (The number of distinct atoms in the model of a m -ary predicate is limited to m^n , so at most m^n times an answer can be added that is not subsumed by the previous ones.). An alternative approach which also ensures termination and completeness of the search discards the $\text{not}(\text{subsumed}(\dots))$ constraints and uses a weaker but easier to verify constraint which restrict the number of answers for a predicate p/m to m^n .

Below, we use \diamond to separate the three components of the state. The symbols **As** and **Bs** stand for any sequence of atoms, while **Abds** stands for a sequence consisting solely of abduce atoms. **Store** stands for a conjunction (set) of constraints, **Answers** for a set of answers (constrained facts) and **Answers_p** for the subset of answers about predicate **p**. The initial state is given by $\text{false} \leftarrow \text{As} \diamond \epsilon \diamond \epsilon$ where **As** is the query. Assume n domain elements. Remember that arguments of program predicates of the abstracted program are always variables.

1. $H \leftarrow \text{Abds}, p(\overline{X}), \text{As} :: \text{Cls} \diamond \text{Answers} \diamond \text{Store}$
 Let $p(\overline{X}_i) \leftarrow \text{Bs}_i$ ($i : 1, \dots, m$) be the clauses in the definition of **p**. If this is the first call to **p** (**Cls** does not contain a clause with an atom $\text{Lookup}(p(\overline{Y}))$) then the new state is :
 $p(\overline{X}_1) \leftarrow \text{Bs}_1 :: \dots :: p(\overline{X}_m) \leftarrow \text{Bs}_m :: H \leftarrow \text{Abds}, \text{Lookup}(p(\overline{X})),$
 $\text{As} :: \text{Cls} \diamond \text{Answers} \diamond \text{Store}$
 Else the new state is:
 $H \leftarrow \text{Abds}, \text{Lookup}(p(\overline{X})), \text{As} :: \text{Cls} \diamond \text{Answers} \diamond \text{Store}$
2. $H \leftarrow \text{Abds}_1, \text{Lookup}(p(\overline{X})), \text{As} :: \text{Cls} \diamond$
 $p(\overline{Y}) \leftarrow \text{Abds}_2 :: \text{Answers} \diamond \text{Store}$
 The new state is:
 $(H \leftarrow \text{Abds}_1, \text{Abds}_2, \text{As})\text{mgu}(\overline{X}, \overline{Y}) :: H \leftarrow \text{Abds}_1, \text{Lookup}(p(\overline{X})),$
 $\text{As} :: \text{Cls} \diamond p(\overline{Y}) \leftarrow \text{Abds}_2 :: \text{Answers} \diamond \text{Store}$
3. $\text{false} \leftarrow \text{Abds} :: \text{Cls} \diamond \text{Answers} \diamond \text{Store}$
 The new state is:
 $\text{Cls} \diamond \text{Answers} \diamond \text{false} \leftarrow \text{Abds} :: \text{Store}$
4. $\text{Cls} \diamond \text{Answers} \diamond \text{Store}$ where **Store** is inconsistent.
Backtrack to the state corresponding to the most recent **choice point** with an untried alternative.

5. $p(\overline{X}) \leftarrow \text{Abds} :: \text{Cls} \diamond \text{Answers} \diamond \text{Store}$
 A **choice point** is created. Under the first alternative the new system is:
 $\text{Cls} \diamond \text{Answers} \diamond \text{subsumed}(p(\overline{X}) \leftarrow \text{Abds}, \text{Answers}_p) :: \text{Store}$
 Under the second alternative the new system is:
 $\text{Cls} \diamond p(\overline{X}) \leftarrow \text{Abds} :: \text{Answers} \diamond$
 $\text{not}(\text{subsumed}(p(\overline{X}) \leftarrow \text{Abds}, \text{Answers}_p)) :: \text{Store}$

Rules 1 and 2 are as before. Rule 3 adds a new constraint to the constraint store and rule 4 checks its consistency. Rule 5 processes a new answer and creates a choice point. The first alternative adds a subsumption constraint and drops the new fact. The second alternative adds a not-subsumed constraint to the store and the answer to the answer set. The not-subsumed constraint is redundant when the subsumed constraint creates an inconsistent store (often the case when a first answer is added). The store is inconsistent when it has more than m^n answers for p/m . So an alternative is to drop the not-subsumed constraints and to check the weaker constraint on the size of $\text{Answers}_{p/m}$. Both cases guarantee termination and completeness of the search for a given domain size.

The efficiency of the consistency checks is crucial for the overall performance. Space lacks to give a full description of the encoding as a finite domain problem. We sketch the main ideas using the even-odd example. Let \mathcal{D} be the domain of the pre-interpretation. We use two kinds of finite domain variables: variables D_t ranging over \mathcal{D} and representing the pre-interpretation of the term t and boolean variables $B_{t_1=t_2}$ indicating whether or not the terms t_1 and t_2 have the same pre-interpretation. Such boolean variables are linked to the domain variables through definitions $B_{t_1=t_2} \leftrightarrow D_{t_1} = D_{t_2}$ which ensure propagation of new information. Which domain variables are created is determined by the terms which occur in the constraints. Consider the constraint $\text{false} \leftarrow \text{abduce}(0, X), \text{abduce}(s(0), X)$. To handle it we introduce domain variables D_0 and $D_{s(0)}$. We can translate the constraint in $\text{false} \leftarrow D_0 = X, D_{s(0)} = X$ or, after elimination of X : $\text{false} \leftarrow B_{0=s(0)}$ or $B_{0=s(0)} = 0$. To express the connection between 0 and $s(0)$, we add for all $d \in \mathcal{D}$ the constraint $B_{0=d} \leq B_{s(0)=s(d)}$ ⁴. Note that this implies the creation of domain variables $D_{s(d)}$. Now consider the constraint $\text{subsumed}(\text{even}(Y)) \leftarrow \text{abduce}(s(s(0)), Y), \{\text{even}(X) \leftarrow \text{abduce}(0, X), \}$. It contains a new term $s(s(0))$, so a domain variable $D_{s(s(0))}$ is created and it is linked with $D_{s(0)}$ by adding for all $d \in \mathcal{D}$ the constraint $B_{s(0)=d} \leq B_{s(s(0))=s(d)}$. The subsumption constraint is expressed as $B_{s(s(0))=0} = 1$ (Its negation as $B_{s(s(0))=0} = 0$). This translation ensures that all choices which are made are immediately propagated.

5 Alternative approaches

Model generation. The logic program and the clause $\text{false} \leftarrow \text{query}$ can be considered as a logical theory. A model of this theory is a proof that the query fails. There exist general purpose tools for generating models of logical theories.

⁴ Or equivalently $B_{0=d} \rightarrow B_{s(0)=s(d)}$.

FINDER [16, 17], written in C, is such a tool, it takes as input a set of clauses in a many-sorted first order language, together with specifications of finite cardinalities of the domains for the sorts, and generates interpretations on the given domains which satisfy all the clauses [16]. The system performs an exhaustive search for interpretations of the given language, using the declared clauses as constraints to direct backtracking. A major difference between FINDER and our approach is that there is no explicit control on the order of evaluation. FINDER generates an interpretation and tests it against all clauses; if all clauses are true with respect to the generated interpretation, we have a model that is accepted and printed. If one or more of the clauses are false, the interpretation is adjusted so as to generate an improved candidate interpretation. The process continues until the search space is exhausted or a model has been found.

Regular approximations. Within the context of program analysis, the most obvious approach to prove failure is to add a clause $\mathbf{shouldfail}(\bar{X}) \leftarrow \mathbf{query}(\bar{X})$ and to use one or another kind of type inference to show that the success-set of $\mathbf{shouldfail}(\bar{X})$ is empty. A typical representative of such systems is described in [11] which computes a regular approximation of the program. Roughly speaking, for each argument of each predicate, the value it can take in the success-set⁵ are approximated by a type (a canonical unary logic program). Failure of the query is proven if the types of $\mathbf{shouldfail}(\bar{X})$ are empty. Also set based analysis [12] can be used to approximate the success-set.

Program specialisation. One could also employ program transformation, and more specifically program specialisation techniques to prove failure of the query. If for the given query, the program can be specialised in the empty program, then the query trivially fails. A technique which has almost the same power as transformations based on the fold/unfold approach is conjunctive partial deduction [13]. By specialising conjunctions of atoms instead of single atoms, it can achieve substantially better results than other specialisers. For example it can specialise the even/odd program into the empty program for our example query.

6 Experiments

For the abductive systems we experimented extensively with 5 different control regimes. We report here on the most successful one i.e. the system with the best worst cases. In this system, as in all others, rule 5, which triggers backtracking, has top priority. Rules 2 and 3, which use respectively tabled answers and abduced facts to perform a resolution step, have equal priority, as well as rule 1 for the case that the clause has a call to an already tabled predicate. If none of these rules apply and there is an unprocessed clause with only calls to not yet tabled program predicate, then rule 1 is applied on that clause. Otherwise rule 4 is applied on a selected clause and the pre-interpretation is extended.

⁵ The set of ground atoms which are logical consequences of the program.

In the constraint system, rule 4 which checks for the consistency of the Store has top priority each time a constraint is added to the store as it can trigger backtracking. Rule 5, which can create a choice point, has lowest priority. A first implementation used the weaker constraints on the number of answers for each predicate instead of the not-subsumed constraints. Using the latter resulted in significant better pruning on the hard problems (and marginal slow-down on easier ones). The finite domain solver is incremental.

Table 2 details the benchmarks⁶. Besides the name, the table gives the domain size, the number of functors, the number of abduced facts in a complete pre-interpretation⁷, the number of predicates defined in the program, the total number of program clauses and the number of program clauses with a head predicate of arity 1, of arity 2 and of arity 3⁸.

`odd_even` is a trivial example about even and odd numbers. `wicked_oe` is an extension which adds a call to each clause and 4 functors which are irrelevant for success or failure. It allows to see the effect of this on the search space. `appendlast`, `reverselast`, `nreverselast` and `schedule` are small but hard examples from the domain of program specialisation. The queries express program properties (which have to be recognised by a specialiser to derive the empty program). `multiset0` and `multiset1` are programs to check the equivalence of two multisets, the first uses a binary operator “o” to build sets, the second uses a list representation. The others are typical examples from a large set of planning problems reasoning on multisets of resources. The first two use the “o” representation for the multiset, the next two the “l” representation. `blockpair2o` and `blockpair2l` omit the for success or failure irrelevant argument for collecting the plan (and have 6 functors less). `blocksol` is there to show what happens when the query does not fail.⁹

The abductive systems are implemented in Prolog. The queries have been executed with Prolog by BIM on a SUN sparc Ultra-2. The constraint system is also written in Prolog, uses the SICSTUS finite domain solver and was running under SICSTUS Prolog [18] on the same machine. FINDER is implemented in C and was running on a IBM RS6000. An example FINDER-program can be found in appendix B. Regular approximations (RA) were computed with a system due to John Gallagher, conjunctive partial deduction (CPD) with a system due to Michael Leuschel. Witold Charatonik was so kind to run our examples on a tool (some info can be found in [5]) for set based analysis (SBA) he developed together with Harald Ganzinger, Christoph Meyer, Andreas Podelski and Christoph Weidenbach.

Table 3 gives the results: For the abductive system the time and the number of backtracks (wrt. the choice made in the pre-interpretation); for the constraint

⁶ The programs are given in appendix A, the code is also available at <http://www.cs.kuleuven.ac.be/~henkv/pre>

⁷ n facts and m domain elements yields $m^n/m!$ different pre-interpretations.

⁸ With arity k, a predicate can have $2^{(m^k)}$ different interpretations.

⁹ One should in parallel run the query and interrupt the search for failure when a solution is found.

name	domain	functor	abduced	predicates	clauses	arity 1	arity 2	arity 3
odd_even	2	2	3	2	3	3	0	0
wicked_oe	2	6	10	3	4	1	3	0
appendlast	3	4	12	2	4	0	2	2
reverselast	3	4	12	2	4	0	2	2
nreverselast	5	4	28	3	6	0	4	2
schedule	3	4	12	6	12	9	3	0
multiset0	2	4	7	1	7	0	7	0
multiset1	2	4	7	2	4	0	2	2
blockpair2o	2	9	19	3	15	0	15	0
blockpair3o	2	15	36	3	15	0	7	8
blockpair2l	2	9	19	5	14	0	8	6
blockpair3l	2	15	36	5	14	0	0	14
blocksol	2	9	19	5	14	0	0	14

Table 2. Properties of benchmark programs

system the time, the number of backtracks (wrt. the choices regarding tabulation) and the number of consistency checks (each check verifies the existence of a pre-interpretation satisfying all constraints and has internal backtracking); for FINDER the time and the number of backtracks. For CPD, RA and SBA we only indicate whether failure of the query follows from the result (It makes little sense to give times as these systems do not perform an exhaustive search and no sense to try **blocksol**). If followed by H, time is in hours, otherwise in seconds.

name	AB		CS			FINDER		CPD	RA	SBA
	time	#bckt	time	#bckt	#cstch	time	#bckt			
odd_even	0.00	4	0.00	0	2	0.02	1	yes	yes	yes
wicked_oe	0.08	64	0.00	0	2	0.02	64	yes	yes	yes
appendlast	6.63	949	0.45	1	6	0.83	19023	yes	no	yes
reverselast	9.37	1267	3.70	2	9	1590	94583354	no	no	yes
nreverselast	>19H	-	>19H	-	-	>15H	> 100.10 ⁶	yes	no	no
schedule	0.11	33	0.31	1	10	0.07	508	no	no	yes
multiset0	0.05	12	0.04	0	6	0.47	2849	no	yes	yes
multiset1	0.01	3	0.06	1	8	77.10	2583088	yes	no	yes
blockpair2o	3.36	103	0.38	0	12	112	1359532	no	no	no
blockpair3o	639.05	97246	0.42	0	12	>1.65H	> 10.10 ⁶	no	no	no
blockpair2l	0.9	34	2.36	2	17	>1.41H	> 10.10 ⁶	no	no	no
blockpair3l	2.46	162	2.49	2	17	>3.80H	> 10.10 ⁶	no	no	no
blocksol	293.33	12832	4.48H	299	609	>1.01H	> 10.10 ⁶	-	-	-

Table 3. Results.

6.1 Discussion

The abductive system is rather sensitive to the presence of functors which are irrelevant to the existence of a solution (but which occur in terms met during top-down execution) as a comparison of `odd_even` and `wicked_oe`, of `blockpair2o` and `blockpair3o` and of `blockpair2l` and `blockpair3l` shows. When larger domain sizes are needed, the size of the search space increases dramatically, so it is hardly a surprise that it fails on `nreverselast`. It did surprisingly well in showing that there is no solution based on a 2-element domain for `blocksol`.

The constraint system is doing consistently well on all planning problems which fail. Its performance is unaffected by the presence of irrelevant functors. It has serious problems with `blocksol` where it backtracks a lot and has to do a big number of consistency checks before having exhausted the search space (not a real problem, one should run the query and find a solution). More serious is the problem it has with `nreverselast` where one needs to distinguish 5 different kinds of lists. The resulting search-space is of the order $5^{28} = 4 * 10^{20}$. The constraints generated by this example, while correct, does not seem to prune the search-space a lot. Adding redundant but stronger constraints will be needed to avoid the current trashing behaviour.

FINDER, which has no equivalent of a top-down strategy and of tabling, is doing poor on these problems. It behaves worse than our first abductive procedure. FINDER's input being sorted, it is possible to associate different sorts with the different functors and different domain sizes with different sorts. Very recently, experimenting with this feature and using the intuitively right domain sizes for the sort "list" (3 for `reverselast` and 5 for `nreverselast`) and 2 for all other types (also for a functor which maps two lists to a pair¹⁰), we succeeded in finding a solution in respectively 0.2s with 2738 backtracks (`reverselast`) and 18H with 111.10^6 backtracks (`nreverselast`). This suggests that separate types and separate domains for different functors could also restrict the search space in our systems. We plan to explore this in the future.

Conjunctive partial deduction can handle some of the problems which are difficult for us (and a planned extension¹¹ likely even more), but cannot handle any of the planning problems. Computing regular approximations is fast, but it can show failure of the most simple problems only. The set based analyser is more precise and fails only on the planning problems and `nreverselast`.

7 Conclusion

For definite logic programs, we have addressed the problem of proving that certain queries cannot succeed with an answer. A problem which is particularly

¹⁰ It is not obvious from the problem formulation that this latter can result in a solution.

But it drastically reduces the search space: with m domain elements, the natural thing is that the sort of pair has m^2 domain elements.

¹¹ A planned extension of CPD should be able to handle also `reverselast` and `multiset`.

relevant when the query does not fail finitely. We have developed two new approaches which aim at searching a model of the program in which the query is false. We have performed some experiments using (rather small) example programs and queries which do not terminate¹². We also did a comparison with other approaches which could be used to tackle this problem: a general purpose model generation tool which does not allow the user to control the search, the use of type inference, and the use of program specialisation. In the case of type inference, the approach is in fact also to compute a model. However, the chosen model is the one which best reflects the type structure of the program. If the query happens to be false in this model, then failure is shown. Also in the case of program specialisation, showing failure is a byproduct of the approach: for some queries, the program happens to be specialised into the empty program.

Abduction is a very powerful and general problem solving technique. It was pretty easy to formulate the problem of searching a pre-interpretation such that the query is false in the least model based on it as an abductive problem and to use a general purpose abductive procedure[9]. But we quickly realised that we had almost no control over the search for a solution. Our first approach was to build a special purpose abductive procedure for definite programs which employs tabulation and which hard wired the constraints that pre-interpretation of functors are total functions. The idea behind the proof procedure is to use a top-down evaluation strategy —abducting a part of the pre-interpretation only when needed in evaluating the query— and to prevent looping by the use of tabulation. Experiments confirmed our intuition that it was important to delay the abduction of new components in the pre-interpretation as long as possible (to propagate all consequences of what was already abduced to check whether it was part of a feasible solution). The system was performing in general much better than FINDER which is at least an order of magnitude faster in raw speed than ours. This suggests that our basic strategy —using a goal directed proof procedure and using tabulation— is a good one. However, the results were not really convincing. We tried to delay the choice of components in the pre-interpretation even more: we considered them as constraints. This allowed to delay the decisions up to the point where answers had to be tabled: at such a point one need to know whether the answer is new or not. Still we do not fix the pre-interpretation at such a point but formulate constraints on the pre-interpretation, using a finite domain solver to check the existence of a pre-interpretation satisfying all constraints. With this approach, the overall speed crucially depends on the speed by which consistency can be checked. We were coding this consistency check as a finite domain problem and obtained quite impressive results. Still the system has some weaknesses. It starts to slow down when it needs a lot of backtracking over its decisions with respect to new answers being subsumed by the existing ones or not. The number of possible backtracks quickly goes up with the arity of the predicates, as the example `blocksol`, where the query does not fail, illustrates. It also increases quickly with the size of the domain needed to show failure e.g.

¹² These programs also loop when using tabulation or when executing bottom-up after a magic set transformation.

`nreverse``last`¹³. Further work is possible to reduce this number of backtracks: tabling one predicate in each strongly connected component of the program is enough to prevent looping of the procedure. This can reduce the number of choice points. Still it is desirable to find ways to extract more knowledge from the problem to further prune the search.

In general, our approach is doing much better than the general purpose model generation tool `FINDER` and type inference based on Regular Approximations. The comparison with conjunctive partial deduction is less straightforward. Both approaches seem to be good at a different classes of problems. In fact it could be interesting to apply program specialisation as a pre-processing step: the specialisation may reduce the number of predicates and the number of functors, making the problem easier to solve by our tool (and in extreme cases making it a trivial problem by returning the empty program).

In a broader context, this paper makes contributions to the following topics:

- A (first) study of methods to prove (infinite) failure of definite logic programs.
- The development of a proof procedure which combines tabulation with abduction and of a constraint based procedure which treats the abducibles as constraints and uses a constraint solver to check the existence of a solution for the abducibles. Also the latter procedure uses tabulation.
- A better understanding of the power and limitations of abduction. While very expressive, our findings suggests that abductive procedures need to be augmented with “background” knowledge to direct the search for abductive solutions. Simply specifying the properties of an abductive solution as an integrity constraint cannot provide sufficient guidance to the search for a solution. It is interesting to observe that background knowledge is also often the key to success in Inductive Logic Programming which makes use of inductive procedures which are in more than one respect “twins” of abductive procedures [1].
- The further development of model based program analysis. [10] showed that model based program analysis pioneered by [6] is also an excellent method for type inference. In [10] it was shown that there exist pre-interpretations which encode various other declarative properties of programs. Our work takes this work one step further by developing methods for automatically constructing a pre-interpretation which expresses a particular program property (expressed as a query which should fail).

A possible extension of this work is for programs with negation. Transformations are known which preserve a 3-valued completion semantics and transform a program with negation into a definite program, having for each predicate p/n two definitions, one for p/n and one for $p */ n$. It seems feasible to apply our method on this transformed program.

¹³ Some problems require an infinite domain, e.g. `less(N,s(N)). less(N,s(M)) <- less(N,M)`. and the query `?- less(N,M), less(M,N) ..`

References

1. H. Ade and M. Denecker. Abductive inductive logic programming. In *Proc. IJCAI'95*, pages 201–1209. Morgan Kaufman, 1995.
2. K.R. Apt, R.N. Bol, and J.W. Klop. On the safe termination of Prolog programs. In *Proc. ICLP'89*, pages 353–368, Lisbon, june 1989. MIT Press.
3. R.N. Bol, K.R. Apt, and J.W. Klop. An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science*, 86(1):35–79, august 1991.
4. M.P. Bonacina and J. Hsiang. On rewrite programs: semantics and relationship with Prolog. *J. Logic Programming*, 14(1&2):155–180, october 1992.
5. W. Charatonik and A. Podelski. Directional type inference for logic programs. In *Proc. SAS'98*, LNCS, Pisa, Italy, 1998. To appear.
6. M. Codish and B. Demoen. Analysing logic programs using “prop”-ositional logic programs and a magic wand. *J. Logic Programming*, 25(3):249–274, December 1995.
7. D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *J. Logic Programming*, 19 & 20:199–260, may/july 1994.
8. D. A. de Waal, M. Denecker, M. Bruynooghe, and M. Thielscher. The generation of pre-interpretations for detecting unsolvable planning problems. In *Proc. IJCAI Workshop on Model based Automated reasoning*, pages 103–112, 1997.
9. M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for abductive logic programs. *J. Logic Programming*, 34(2):201–226, Februari 1998.
10. J. Gallagher, D. Boulanger, and H. Sağlam. Practical model-based static analysis for definite logic programs. In *Proc. ILPS'95*, pages 351–365, Portland, Oregon, December 1995. MIT Press.
11. J. P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs,. In *Proc. ICLP'94*, pages 599–613. MIT Press, 1994.
12. N. Heintze. *Set based program analysis*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.
13. Michael Leuschel, Danny De Schreye, and André de Waal. A conceptual embedding of folding into partial deduction: Towards a maximal integration. In *Proc. JICSLP'96*, pages 319–332, Bonn, Germany, September 1996. MIT Press.
14. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. SIGMOD 1994 Conf. ACM*. Acm Press, 1994.
15. Y. D. Shen. An extended variant of atoms loop check for positive logic programs. *New Generation Computing*, 15(2):187–203, 1997.
16. J. Slaney. Finder: Finite domain enumerator system description. Technical Report TR-ARP-2-94, Centre for Information Science Research, Australian National University, Australia, 1994. Also in Proc. CADE-12.
17. J. Slaney. Finder - finite domain enumerator - version 3.0 - notes and guide. Technical report, Centre for Information Science Research, Australian National University, Australia, 1997.
18. Swedish Institute of Computer Science. *SICSTUS Prolog User's Manual*, november 1997.
19. H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proceedings ICLP'86*, LNCS, pages 84–98, London, 1986. Springer-Verlag.
20. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT press, 1989.
21. David S. Warren. Memoing for Logic Programs. *Communications of the ACM*, 35(3):93–111, March 1992.

Appendix A: example programs

odd_even. The even/odd program. Here and further on, the considered queries are the bodies of 0-arity predicates. The query `odd_even` has no answer.

```
odd_even:- even(X), even(s(X)).
```

```
even(zero).
even(s(X)):- odd(X).
odd(s(X)):- even(X).
```

wicked_oe. A version of even odd with an extra superfluous argument which creates a term with 4 different functors:

```
wicked_oe:- weird_even(X, _U1), weird_even(s(X), _U2).
```

```
weird_even(zero,U) :- weird_p(U).
weird_even(s(X),U) :- weird_odd(X,_V), weird_p(U).
weird_odd(s(X),U) :- weird_even(X,_V), weird_p(U).
weird_p(f(g(h(a))))).
```

appendlast. The `appendlast`-query first appends the list `[a]` to a undefined list becoming the list `Xs`. The subsequent call to `last` can never succeed as `b` can never be the last element of the list `Xs`.

```
appendlast:- app(X, [a], Xs), last(Xs, b).
```

```
app([],L,L).
app([H|X],Y,[H|Z]) :- app(X,Y,Z).
last([X],X).
last([H,H2|T],X) :- last([H2|T],X).
```

reverselast. The query `reverselast` is similar to the query `appendlast` but uses `reverse` with accumulator to put `[a]` at the end of the list. Predicates defined in examples above are not repeated, e.g `last`.

```
reverselast:- reva(L, R, [a]), last(R, b).
```

```
reva([], Acc, Acc).
reva([Y|Z], R, Acc):-
    reva(Z, R, [Y|Acc]).
```

nreverselast. The query `nreverselast` uses naive `reverse` to put the element `a` at the end of the list.

```
nreverselast :- rev([a|X], R), last(R, b).
```

```
rev([], []).
```

```

rev([X|Y], R):-
    rev(Y, S),
    app(S, [X], R).

```

schedule. The `schedule` problem is example with a more complicate reason for failure.

```

schedule:- cFirst(R), mv(R).

```

```

mv(R):- tr(R,NewR), mv(NewR).
mv(R):- atleast2c(R). % success iff R is non-safe state
tr([c,n|Rs], [n,c|Rs]).
tr([n|Rs], [n|NewRs]):- tr(Rs,NewRs).
tr([], []).
cFirst([c|Qs]):- nOnly(Qs).
nOnly([n|Qs]):- nOnly(Qs).
nOnly([n]).
atleast2c([c|L]):- atleast1c(L).
atleast2c([n|L]):- atleast2c(L).
atleast1c([c|_]).
atleast1c([n|L]):- atleast1c(L).

```

multiset. `sameMultiSet` is a predicate to check that two multisets contain the same elements. The multiset is represented with a functor `o/2` and a constant `emptyMultiSet`. Without tabulation, it also has an infinite search tree for ground queries.

```

multiset0 :- sameMultiSet(o(a,o(a,emptyMultiSet)),
                        o(_X,o(emptyMultiSet,b))).

```

```

sameMultiSet(X, X).
sameMultiSet(o(X, Y), o(X, Z)):-
    sameMultiSet(Y, Z).
sameMultiSet(o(o(X, Y), Z), U):-
    sameMultiSet(o(X, o(Y, Z)), U).
sameMultiSet(U, o(o(X, Y), Z)):-
    sameMultiSet(U, o(X, o(Y, Z))).
sameMultiSet(o(emptyMultiSet, X), Y):-
    sameMultiSet(X, Y).
sameMultiSet(X, o(emptyMultiSet, Y)):-
    sameMultiSet(X, Y).
sameMultiSet(o(X, Y), Z):-
    sameMultiSet(o(Y, X), Z).

```

multisetl. The same problem can be writing using lists. The query does not terminate due to the presence of the variable.

```

multiset1 :- sml([a], X), sml(X, [b]).

sml([], []).
sml([X|Y], D) :- delete(X, D, E), sml(Y, E).
delete(M, [M|T], T).
delete(M, [H|T], [H|L]) :- delete(M, T, L).

```

blockpair??. A simple planner in the blocks-world due to Michael Thielscher. It has to be executed under iterative deepening to find plans for most problems (which have a solution). We have two different action theories from the blocks world. Blocks are identified by integers. The action theory, `actionPair`, has, besides the usual actions of the blocks world, an action to add or remove a pair of blocks. We have variants with two arguments (which do not collect the plan) and with three argument (the plan is stored in the second argument). And we have versions using multisets based on the functor `o/2` and based on lists. For the latter, the names end on *l* e.g. `causesPair1/2` is the 2 argument version using lists and the `actionPair` theory. For the `actionPair` theory, the unsolvable query has an initial state with an even number of blocks and a final theory with an odd number of blocks. The query `blocksol` has a solution (can be found using iterative deepening).

```

blockpair2o:-
    causesPair(o(on(s(nul),nul),o(ta(nul),o(cl(s(nul)),em))),
               o(on(s(s(nul)),s(nul)),o(on(s(nul),nul),
               o(ta(nul), o(cl(s(s(nul))),em)))).

blockpair3o:-
    causesPair(o(on(s(nul),nul),o(ta(nul),o(cl(s(nul)),em))),
               _Plan,
               o(on(s(s(nul)),s(nul)), o(on(s(nul),nul),
               o(ta(nul), o(cl(s(s(nul))),em)))).

blockpair2l:-
    causesPair1([on(s(nul),nul),ta(nul),cl(s(nul)),em],
                Sequence),
    m_subset([on(s(s(nul)),s(nul)),on(s(nul),nul),ta(nul),
                cl(s(s(nul))),em],
                Sequence, []).

blockpair3l:-
    causesPair1([on(s(nul),nul),ta(nul),cl(s(nul)),em],
                _Plan, Sequence),
    m_subset([on(s(s(nul)),s(nul)),on(s(nul),nul),ta(nul),
                cl(s(s(nul))),em],
                Sequence, []).

blocksol:- causesZero1([on(s(nul),nul),ta(nul),cl(s(nul)),em],
                       [on(s(s(nul)),s(nul)),on(s(nul),nul),ta(nul),
                       cl(s(s(nul))),em]).

```

```

causesPair(I1, I2):-
    sameMultiSet(I1, I2).
causesPair(I, G):-
    actionPair(C, E),
    sameMultiSet(o(C, Z), I),
    causesPair(o(E, Z), G).

actionPair(ho(V), o(ta(V),o(cl(V),em))).
actionPair(o(cl(V),o(ta(V),em)), ho(V)).
actionPair(o(ho(V),cl(W)), o(on(V,W),o(cl(V),em))).
actionPair(o(cl(V),o(on(V,W),em)), o(ho(V),cl(W))).
actionPair(o(on(V,W),o(cl(V),em)),
            o(on(s(s(V)),s(V)), o(on(s(V),V), o(on(V,W),
            o(cl(s(s(V))),em)))).
actionPair(o(on(s(s(V)),s(V)), o(on(s(V),V), o(on(V,W),
            o(cl(s(s(V))),em))),
            o(on(V,W), o(cl(V),em))).

causesPair(I1, void, I2):-
    sameMultiSet(I1, I2).
causesPair(I, plan(A, P), G):-
    actionPair(C, A, E),
    sameMultiSet(o(C, Z), I),
    causesPair(o(E, Z), P, G).

actionPair(ho(V), put_down(V), o(ta(V),o(cl(V),em))).
actionPair(o(cl(V),o(ta(V),em)), pick_up(V), ho(V)).
actionPair(o(ho(V),cl(W)), stack(V,W), o(on(V,W),o(cl(V),em))).
actionPair(o(cl(V),o(on(V,W),em)), unstack(V), o(ho(V),cl(W))).
actionPair(o(on(V,W),o(cl(V),em)), add_two,
            o(on(s(s(V)),s(V)), o(on(s(V),V), o(on(V,W),
            o(cl(s(s(V))),em)))).
actionPair(o(on(s(s(V)),s(V)), o(on(s(V),V), o(on(V,W),
            o(cl(s(s(V))),em))),
            delete_two,
            o(on(V,W), o(cl(V),em))).

causesPairl(I, void, I).
causesPairl(I, plan(A, P), G) :-
    actionPairl(C, A, E),
    m_subset(C, I, Z),
    app(E, Z, S),
    causesPairl(S, P, G).

```

```

actionPair1([ho(V)],put_down(V),[ta(V),cl(V),em]).
actionPair1([cl(V),ta(V),em],pick_up(V),[ho(V)]).
actionPair1([ho(V),cl(W)],stack(V,W),[on(V,W),cl(V),em]).
actionPair1([cl(V),on(V,W),em],unstack(V),[ho(V),cl(W)]).
actionPair1([on(V,W),cl(V),em],add_two,
            [on(s(s(V)),s(V)),on(s(V),V),on(V,W),cl(s(s(V))),em]).
actionPair1([on(s(s(V)),s(V)),on(s(V),V),on(V,W),cl(s(s(V))),em],
            delete_two,[on(V,W),cl(V),em]).

```

```

m_subset([], L, L).
m_subset([H|T], L1, L2):-
    delete(H, L1, L3),
    m_subset(T, L3, L2).

```

```

causesPair1(I,I).
causesPair1(I,G):-
    actionPair1(C,E),
    m_subset(C,I,Z),
    app(E,Z,S),
    causesPair1(S,G).

```

```

actionPair1([ho(V)], [ta(V),cl(V),em]).
actionPair1([cl(V),ta(V),em], [ho(V)]).
actionPair1([ho(V),cl(W)], [on(V,W),cl(V),em]).
actionPair1([cl(V),on(V,W),em], [ho(V),cl(W)]).
actionPair1([on(V,W),cl(V),em],
            [on(s(s(V)),s(V)),on(s(V),V),on(V,W),cl(s(s(V))),em]).
actionPair1([on(s(s(V)),s(V)),on(s(V),V),on(V,W),cl(s(s(V))),em],
            [on(V,W),cl(V),em]).

```

```

causesZero1(I, I).
causesZero1(I, G):-
    actionZero1(C, E),
    m_subset(C, I, Z),
    app(E, Z, S),
    causesZero1(S, G).

```

```

actionZero1([ho(V)], [ta(V), cl(V), em]).
actionZero1([cl(V), ta(V), em], [ho(V)]).
actionZero1([ho(V), cl(W)], [on(V,W), cl(V), em]).
actionZero1([cl(V), on(V, W), em], [ho(V), cl(W)]).
actionZero1([on(X, Y), cl(X), em],
            [on(s(X), X), on(X, Y), cl(s(X)), em]).

```

Appendix B: an example of input for FINDER

FINDER supports only nulladic, monadic and dyadic functions. As predicates are declared as functions with **boolean** as value sort, these restrictions also apply to predicate definitions and forces some transformation for predicates with arity > 2 : for example a ternary atom $p(t_1, t_2, t_3)$ is encoded as $p(t_1, \text{arg}_p(t_2, t_3))$ where $\text{arg}_p/2$ is a binary functor used to encode atoms with predicate $p/3$ and which constructs a term in a sort different from the sort of the terms t_1, t_2 , and t_3 of the original atom. To have the same expressivity as the original p based on a 2 element domain pre-interpretation, the pre-interpretation of arg_p has to be based on a 4 element domain.

Input for the **multiset1** problem. The ternary predicate **delete/3** has been converted in a binary **delete_was3/2** predicate. Besides the sort **term** for all terms of the original program, a sort **pair** has been introduced. The extra functor **delete_argpair** is used to bundle two terms of sort **term** into one of sort **pair**.

```
sort { term cardinality = 2.
      pair cardinality = 4.
    }

const {a: term.
      b: term.
      nil: term.
      ml1: bool.
    }

function { cons: term, term -> term.
          delete_argpair: term, term -> pair.
          delete_was3: term, pair -> bool.
          sml: term, term -> bool.
        }

clause {ml1 -> false.
       sml(cons(a, nil), z), sml(z, cons(b, nil)) -> ml1.
       sml(nil, nil).
       delete_was3(y, delete_argpair(w, z)), sml(x, z)
         -> sml(cons(y, x), w).
       delete_was3(z, delete_argpair(cons(z, y), y)).
       delete_was3(w, delete_argpair(x, z))
         -> delete_was3(w, delete_argpair(cons(y, x), cons(y, z))).
    }

setting {solutions: 1
        verbosity {models: brief    stats: full    job: brief
    }
```