Towards a more declarative language for solving finite domain problems

Maurice Bruynooghe, Nikolay Pelov, Marc Denecker

Departement Computerwetenschappen, Celestijnenlaan 200A, Katholieke Universiteit Leuven, B-3001 Heverlee, Belgium
{maurice, pelov, marcd}@cs.kuleuven.ac.be,
tel. +32 16 327539, fax +32 16 327996

Abstract. It has been observed that the use of an abductive formalism allows to formulate finite domain problems in a much more declarative style than with the use of a finite domain constraint language. However, such a system tends to have complex inference rules and it is very difficult to understand how the formalization influences the performance of the abductive solver.
This paper explores the expressive power and the computational requirements for a much more restrictive abductive system where the abductive component is restricted to the abduction of open function symbols. The result is a language and an evaluation mechanism which remains more close to conventional logic programming than other abductive systems

1 Introduction

Many constraint programs solving a finite domain problem are written in a rather procedural style: setting up a data structure containing the domain variables, creating constraints and then enumerating the domain variables. A typical example is the program solving the n-queens problem:

```
queens(Q,N) :-
   generate(Q,N,N),
   safe(Q),
   instantiate(Q).

generate([],0,_).
generate([X|T],M,N) :- M > 0, X in 1..N, M1 is M - 1,
                       generate(T,M1,N).

safe([]).
safe([X|T]) :- noAttack(X,1,T),safe(T).

noAttack(_,_,[]).
noAttack(X,N,[Y|Z]) :-
        X \= Y,
```

```
        Y \= X + N,
        X \= Y + N,
        S is N + 1,
        noAttack(X,S,Z).

instantiate([]).
instantiate([X|T]) :-
        enum(X),
        instantiate(T).
```

With such an approach, the solution is obtained as a list of numbers where the i^{th} number indicates the column occupied by the queen in row i. Much more comprehensive and declarative would be to present the result as the least model of a predicate *position/2* where *position(i,j)* represents that the queen in row i is placed on column j. Having available a predicate *position/2* would also allow for a much more declarative style of stating the constraints. For example, the least model of the *position/2* predicate must satisfy the following axiom (we use a Prolog-like syntax for the right-hand-side with ";" for disjunction):

```
% a queen on row R1 should not attack a queen on a higher row R2
false <- position(R1,C1), position(R2,C2), R1<R2,
         (C1=C2 ; R2-R1=C2-C1 ; R2-R1=C1-C2).
```

Using a general purpose abductive system such as SLDNFA [2], one can state that *position/2* is an open predicate, formulate the constraints on the open predicate as integrity constraints, and ask for an abductive solution of the problem satisfying all integrity constraints. However SLDNFA is very general and its performance is no match for dedicated CLP systems. The situation can be alleviated by integrating abductive systems with CLP systems: using the abductive system to generate the constraints to be solved by the CLP system. This approach is taken in [6] and [9]. In this paper we explore a slightly different approach by severely restricting the use of abduction. However we retain an expressive language. Moreover, the execution mechanism remains more close to the familiar SLD or SLDNF procedure and hence is easier to understand and to control.

The core component of our approach is the introduction of open functions. Their introduction allows to eliminate the abduction of atomic facts while retaining much of the expressivity of abductive systems. The execution mechanism becomes almost that of SLDNF. The only extra difficulty is the presence of open function symbols for which standard unification cannot be applied. Their processing gives rise to residual constraints which can be handed over to a solver.

For the queens problem, introducing an open function *pos/1*, one can define the predicate *position/2* as:

```
position(X,pos(X)) :- size(N), X in 1..N.
```

where *size/1* defines the size of the board and *X in Y..Z* defines the integer values X in the interval Y to Z.

In section 2, we introduce open functions and define a semantics for our logic. Section 3 explores the relationship with abduction. In section 4 we use some examples to discuss the general principles of the execution mechanism. We show that it is feasible and interesting to integrate tabulation. In section 5 we show some more examples of finite domain constraint programming problems for which abductive formulations have been worked out for other abductive systems. Section 6 gives a more formal presentation of a simple proof procedure for our language. We finish with a discussion in Section 7.

2 The language and its semantics

In the logic that we define, a theory is built from three sorts of expressions:

- declarations of open functions:

  $$open\_function(f/n)$$

  Also constants (n=0) can be open functions. A term with an open function as principal functor is called an *open term*.
- Rules defining predicates. They have the same form as in standard logic programming:

  $$A \texttt{:-} L_1, \ldots, L_n$$

  where A is an atom and $L_1, \ldots, L_n$ are literals.
- Integrity constraints. While from a semantic point of view any FOL axiom could be allowed, we constrain integrity rules to be in the form

  $$A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n$$

  with A_i atoms and B_j literals. As will become clear, $A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n$ is semantically and computationally equivalent to $\leftarrow B_1, \ldots, B_n, not(A_1), \ldots, not(A_m)$. We allow for a non-empty disjunction in the head because it gives often rise to a more readable formula. The restriction to clausal form is motivated by the need to give the programmer control over the evaluation of the integrity constraints (left to right execution order).

Note that we use different connectors to distinguish rules (`:-`) from integrity constraints ($\leftarrow$). In the examples we also make use of ";" to denote disjunctions in the right-hand-side of rules and integrity constraints as is common in Prolog.

A fourth component in the language has no direct bearing on the semantics but is a directive stating for which predicates a model satisfying all integrity constraints has to be computed. It is of the form

$$? - model(L)$$

with L either a predicate name or a list of predicate names. In the sequel, we refer to these predicates as the *target* predicates.

A theory in the logic consists of expressions of the above sort. It represents an axiomatization of the problem domain. Intuitively, the meaning of the theory is as follows:

- The domain of discourse is the Herbrand universe of all function symbols and constants that are not open functions. These are called the free function symbols and constants. The alphabet consisting of these symbols is denoted Σ_f. Its Herbrand universe is denoted $HU(\Sigma_f)$.
  In extensions of the logic with CLP-domains, the domain of discourse will be extended with the elements of the CLP domain, e.g. the set of reals, rationals or integers.
- The open functions represent arbitrary functions in the domain of discourse.
- The set of rules forms an inductive definition, defining all user defined predicates in terms of equality, CLP predicates and the free and open function symbols.
  For example, in the queens problem, Example 1 in Section 4, the defined predicate *size/1* depends only on the free function symbols while *position/2* also depends on the open function symbols.
- The role of integrity constraints in this logic is the same as the role of FOL axioms in classical logic. They represent properties of the problem domain. On the level of the semantics, they have the same role as in classical logic: they filter out all undesirable interpretations.

The set of rules in a theory represents an (inductive) definition of the user-defined predicates. As argued in [1], the well-founded semantics is a generalized principle of inductive definitions with negation. Consequently, the semantics of the logic defined here is a simple integration of an extension of the well-founded semantics for logic programming and classical logic semantics for the integrity constraints.

A 3-valued interpretation I is a model of a theory T iff

- Its domain of discourse D_I is $HU(\Sigma_f)$. (Hence, open functions are interpreted as functions within this domain.)
- I satisfies all integrity constraints IC in T. That is, each integrity constraint $IC \in T$ has truth value *true* in I. As usual, this is denoted by $I \models IC$.
- I is a well-founded model of the set of rules in T, as explained below.

In the theory of logic programming, given a predicate logic program P in an alphabet Σ, its well-founded model is defined as the well-founded model of the propositional grounding of P in the Herbrand universe $HU(\Sigma)$.

This concept of well-founded model can be easily extended to the case of rules with open functions. Let R be the set of rules in our theory and I some pre-interpretation of the theory with domain $HU(\Sigma_f)$. The grounding of R w.r.t. I is obtained as follows. Consider first the grounding S_g of R w.r.t. the full alphabet Σ with the open functions. Then the I-grounding of R is obtained by evaluating all terms in S_g w.r.t. I. That is: all terms occurring in S_g are replaced by their interpretation under I.

The result is a set of ground rules in the Herbrand base of Σ_f. I is a well-founded model of R iff I is a well-founded model of the I-grounding of R.

Note that the integrity constraints contribute to Σ_f but have otherwise no influence on the well-foundedness of a model. Their role is in condition 2, to

reject or accept a well-founded model based on a particular interpretation of the open functions.

A special case. When all rules in R are definite rules, i.e. do not contain negative literals, then the complex notion of well-founded model can be replaced by the much simpler notion of least model. In that case, an interpretation I is a model of T iff I satisfies the two first conditions in the above definition and is the least model of R extending the pre-interpretation in I.

3 Relation with abductive reasoning

In the n-queens problem mentioned in the introduction, the key computational step in finding a solution consists of finding the right interpretation for the open functions. This is typical for problems we address with our system. It is a kind of abductive problem and can formally be stated as follows:

Definition 1. *Given a logic theory T with open function declarations, rules and integrity constraints, the abductive problem is to find a pre-interpretation I that can be extended to a model of T.*

This definition is both related to abduction and model generation. First, this problem formulation is remarkably simple compared to more traditional definitions of abduction in the context of logic programming, e.g. in [5, 2]. In traditional treatments, abduction is defined as the search for a set Δ of ground abducible atoms and a substitution θ such that for a given query $\leftarrow Q$, the set of rules in the program plus the atoms Δ is consistent, entails the FOL axioms and entails the universal closure $\forall(\theta(Q))$.

The latter condition means that θ is an answer substitution for the free variables of Q, given Δ.

The relationship with the definition above is stronger than may seem:

- The interpretation of the open functions can be made explicit by means of an open predicate *map/2*. Open terms t can be replaced by a fresh variable X and an atom $map(t, X)$ can be added to the right-hand-side of rules and integrity constraints. Then the problem consists of finding an abductive solution Δ for the open predicate *map/2*.
- In the above problem formulation there is no query to be explained. However, a query can be solved by introducing open constants for the variables of interest, a new predicate and an integrity rule linking the query with the new predicate. More precisely, given an atomic query q with free variables $X_1, \ldots, X_n$, open constants $a_1, \ldots, a_n$ are introduced. Let q' be the query with the variables replaced by the open constants. Then we add the definition $answer(a_1, \ldots, a_n).$ and the integrity constraint $q' \leftarrow$ and query for the model of $answer/n$. By doing so our logic in effect computes an assignment σ for these new open constants such that $\sigma(q)$ is entailed by the computed pre-interpretation. This is (almost) equivalent with the computation of an answer substitution.

The sort of abduction as defined above, is also very related to model generation. Indeed, note that each pre-interpretation I can be extended in at most one way to a model of T. This is because given the pre-interpretation, the set of rules of T allows only one well-founded model. In this sense, the problem of computing I is equivalent with computing a model of T.

Hence, our problem is on the borderline between abduction and model generation.

4 Execution mechanism

Example 1. The queen problem can be formulated as follows:

```
open_function(pos/1).
size(8).
position(X,pos(X)) :- size(N),  X in 1..N.

% column positions are on the board
Y in 1..N <- size(N), position(X,Y).

% a queen on row R1 should not attack a queen on a higher row R2
false <- position(R1,C1), position(R2,C2), R1<R2,
         (C1=C2 ; R2-R1=C2-C1 ; R2-R1=C1-C2).

% query
?- model(position/2).
```

A straightforward execution strategy is to write the integrity constraints as denials and to evaluate them using SLDNF. However, unification has to take into account that the interpretation of the open functions is not the Herbrand interpretation but an unknown one. Hence the equalities involving open function symbols should not be solved using standard unification but give rise to residual equalities. To satisfy the integrity constraints, the interpretation of the open functions has to be such that the residual constraints reduce to false. The search for such an interpretation can be tackled using a finite domain constraint solver.

For example, the denial corresponding to the first integrity constraint is $\leftarrow size(N), position(X,Y), not(Y\ in\ 1..N)$. After some SLD steps, the denial $\leftarrow not(pos(1)\ in\ 1..8)$ is obtained. Evaluating the query $\leftarrow pos(1)\ in\ 1..8$ gives rise to conditional solutions $pos(1) = 1, \ldots, pos(1) = 8$. Hence, $not(pos(1)\ in\ 1..8))$ fails if $pos(1) = 1 \vee \ldots \vee pos(1) = 8$. Evaluating the other branches of the search tree yields similar constraints $pos(i) = 1 \vee \ldots \vee pos(i) = 8$ for all i in the range 1 to 8. Each of these constraints defines the range of the interpretation of the term $pos(i)$ as a finite domain. In a similar way, the other integrity constraint can also be reduced to residual equality and disequality constraints involving the open function *pos/1*. The whole set of constraints can be handed over to a finite domain solver which then returns an interpretation satisfying all constraints.

This interpretation can then be used to compute the least model of the target predicate *position/2*.

A source of inefficiency is that the target predicate(s) are evaluated several times. The use of tabulation [8, 10, 7] can be considered to reduce this overhead. When the open function symbols do not occur during the evaluation of the bodies of the definition of the target predicates –a sufficient condition can easily be verified syntactically– then these bodies can be directly executed under SLDNF, without monitoring the unifications. The answers –with uninterpreted open function symbols– can be tabled as facts to be used by all later calls to the target predicates. This approach is valid for most of the examples we have considered so far and could substantially reduce the computational cost.

A necessary condition for obtaining a solution is that the evaluation terminates. By using a left to right evaluation strategy for bodies of rules and integrity constraints, we give the programmer some control. The situation is similar to that of Prolog programming. The efficiency and termination of the evaluation relies on the careful ordering of bodies by the programmer. The main difference, which makes termination harder to achieve than in logic programming, is the presence of the open function symbols. Indeed, unification with open function symbols never fails as it is delayed. The effect on termination properties is close to the effect of replacing open terms by variables.

4.1 Tabulation

As we have seen above, it is rather straightforward to integrate tabulation in our evaluation mechanism. This is because the evaluation strategy remains very close to that of SLDNF. Besides enhancing performance, tabulation can, as in logic programming, be essential to ensure termination of the evaluation. As an example, consider the problem of finding a closed path in a graph which visits each node exactly once.

Example 2. Hamiltonian path.

```
open_function(path/1)
edge(1,2).
edge(2,3).
edge(1,3).
edge(3,4).
edge(3,1).
edge(4,1).

node(X) :- edge(X,Y).
node(X) :- edge(Y,X).

path(X,path(X)) :- node(X).

reachable(X,Y) :- path(X,Y).
```

```
reachable(X,Y) :- path(X,Z), reachable(Z,Y).

% constraints

% path is following the edges
edge(X,Y) <- path(X,Y).

% all nodes are reachable from node 1
reachable(1,X) <- node(X).

% redundant constraints which can be useful to prune the search space
% no node is connected to itself (assuming more than one node)
false <- path(X,X).

% each node has one incoming path
X=Y <- path(X,Z), path(Y,Z).

% query
?- model(path/2).
```

One of the integrity constraints depend on the defined predicate *reachable/2* which is recursive. Evaluation of this integrity constraints under SLDNF is non-terminating. Tabling the *reachable/2* predicate avoids the loop. The presence of open function symbols makes that we have to do with tabulation in the context of constraint logic programming as e.g. in [4]. This implies some modifications with respect to standard tabulation. Firstly, —specific to our use of open functions— it is inefficient to table two calls which only differ in a open term as unifications which such a term have to be delayed anyway. Better is to replace that term with a fresh variable as this reduces the number of different call patterns. The unifications involving the open function symbol are then introduced by matching the answers against the original call pattern. Secondly, —as with tabulation in the context of constraint logic programming— answers to tabled predicates are constrained facts, the constraints being equalities involving the open function symbols. Hence the lookup operation which uses tabled answers to solve a call has to cope with constrained facts instead of simple atomic facts. Finally, in general it may happen that a call gives rise to syntactically distinct answers which are semantically equivalent. A subsumption tests which cares about the residual constraints may reduce the total number of answers and hence the overall cost of the computation.

Our prototype has tabling incorporated and is able to solve this problem. Other abductive systems such as SLDNFA [2] are trapped in a loop by the recursive definition of reachable.

5 Some more examples

5.1 Job-shop problem

Example 3. Job-shop scheduling is defined as the problem of assigning in time n jobs on m machines where each job is a request for the scheduling of a set of tasks with a particular order. In our formalism it can be described as:

```
open_function(start/1)
% a database with facts
% job(name,deadline,releasetime).    % name is a key
% task(name,job,duration,resource).  % name is a key
% follows(t1,t2).                    % task t1 has to starts after end of t2

start(T,start(T)) :- task(T,J,D,R).

% definition of two tasks overlapping in time
overlap(T1,T2) :- start(T1,S1), task(T1,J1,D1,R1),
                  start(T2,S2), task(T2,J2,D2,R2),
                  overlapping_interval(int(S1,S1+D1-1),int(S2,S2+D2-1)).

overlapping_interval(int(S1,E1),int(S2,E2)) :- S1<=S2, E1>=S2.
overlapping_interval(int(S1,E1),int(S2,E2)) :- S2<S1, E2>=S1.

% tasks are completed before the deadline of a job
S+D-1 <= Dln  <- start(T,S), task(T,J,D,R), job(J,Dln,Rlt).

% tasks do not start before the releasetime of a job
S >= Rlt <- start(T,S), task(T,J,D,R), job(J,Dln,Rlt).

% tasks using the same resource do not overlap
T1=T2  <-  task(T1,J1,D1,R), task(T2,J2,D2,R), overlap(T1,T2).

% tasks respect the ''follows'' relation
S1 >= S2+D2  <- follows(T1,T2), start(T2,S2), task(T2,J2,D2,R2), start(T1,S1).

?- model(start/2).
```

The evaluation of the integrity constraints sets up all constraints. The finite domain solver then searches for a solution.

5.2 Power plant maintenance

Example 4. A fragment of a power plant maintenance problem borrowed from [3] where it is described within SLDNFA.

```
open_function(start/3).
```

```
% a database with facts
% peak(w,p) % expected use in weak w is p
% capacity(p,u,c) % unit u of plant p has capacity of c
% duration(p,u,m,d) % maintenance m of unit u in plant p takes d weeks

maintenance(P,U,M,start(P,U,M)) :- duration(P,U,M). % the start time

% the capacity when all plants are operational
maxcap(M) :- findall(C,capacity(P,U,C),L), sum(L,M).

% definition of two maintenances overlapping in time
overlap(P1,U1,M1,P2,U2,M2) :-
        maintenance(P1,U1,M1,S1), maintenance(P2,U2,M2,S2),
        duration(P1,U1,M1,D1), duration(P2,U2,M2,D2),
        overlapping_interval(int(S1,S1+D1-1),int(S2,S2+D2-1)).

overlapping_interval(int(S1,E1),int(S2,E2)) :- S1<=S2, E1>=S2.
overlapping_interval(int(S1,E1),int(S2,E2)) :- S2<S1, E2>=S1.

% definition of unit U of Plant P with capacity C being on maintenance
% during week W
onmaintenance(P,U,W,C) :- maintenance(P,U,M,S), duration(P,U,M,D),
        overlapping_interval(int(S,S+D-1,int(W,W)), capacity(P,U,C).

% definition of reserve capacity R during week W
reservecapacity(W,R) :- maxcap(Max), peak(W,Pk),
        findall(C,onmaintenace(P,U,W,C),L), sum(L,S),
        R=Max-S-Pk.

% S is sum of elements in L
sum(L,S) :- ... %  omitted

% reserve capacity during a week should be greater than 10000
R > 10000 <- W in 1..52, reservecapacity(W,R).

?- model(maintenance/4).
```

findall(C,capacity(P,U,C),L), sum(L,M) is the Prolog code which computes the sum of the capacities of all units. (Expressing the declarative meaning of M requires the use of second order logic [3]). The use of the *findall/3* predicate in the definition of *maxcap/1* is not a problem as this definition does not involve open functions. However, the definition of *onmaintenace/4* depends on the open functions, hence it has constrained answers. It implies the execution of the *findall/3* predicate (and the calls following it) has to be delayed until the solver has chosen an interpretation for the open functions. This is a situation where the generation of constraints and the solving have to be tightly integrated and

where the choices made by the solver can force backtracking over the processing of some of the integrity constraints. This contrast with previous examples where the integrity constraints could be fully evaluated before the solver solves the constraints involving the open function symbols. This example is outside the scope of our current prototype implementation.

5.3 Planning

One of the strengths of abductive systems such as SLDNFA is in the declarative expression of planning problems in, for example, the event calculus. To construct a plan, some unknown number of facts *action(E,A)* are to be abduced to reach a desired goal state. In such a fact, E is the sequence number of the action and A is a term describing the action such as e.g. *pick_up(b1)*. However in practice it is necessary to limit the number of actions in order to guarantee termination in case no plan exists. At first glance this problem seems beyond the expressivity of our language. However, the following fragment shows how we can cope with the open predicate *action/2*. The approach is to distribute the information in the term describing the action over a number of binary relations. In this fragment they are *name_of_action/2, source_of_action/2* and *destination_of_action/2*.

```
open_function([action/1,source/1,destination/1]).
% the upper bound on the number of actions
bound(20).
% the blocks of our blocks world
block(b1).   ...     block(bn).
% definition of the name_of_action 2 predicate
name_of_action(X,action(X)) :- bound(N), X in 1..N.
% for actions involving parameters such as pick_up/1, a predicate
% source_of_action/2 defines the source block involved in the action.
source_of_action(X,block(X)) :- name_of_action(X,Y), (Y=pick_up; Y=put_down).
% put also has a detination
destination_of_action(X,block(X)) :- name_of_action(X,Y), Y=put_down.
% integrity constraints
% the domain of the open function action/1
Y=put_down;Y=pick_up,Y=skip <- name_of_action(X,Y)
% the domain of the open function source_of_action/1
block(Y) <- source_of_action(X,Y).
% the domain of the open function destination_of_action/1
block(Y),Y=table <- destination_of_action(X,Y).
% the target predicates
?- model([name_of_action/2,source_of_action/2,destination_of_action/2]).
```

The use of the skip action allows to search for a plan of exactly 20 steps. Because actions have varying number of parameters, the definition of target predicates such as *source_of_action/2* depends on another target predicate. This introduces extra difficulty for the evaluation mechanism and is outside the scope of our

current prototype. However, the fragment shows that it is possible to describe the problem without introducing too much extra complexity in the formalization.

6 A simple proof procedure

In this section we present more formally a simple proof procedure for evaluating programs in our language.

A *state* is a tuple $\langle G \,||\, C \rangle$ where G is a normal goal and C is a formula representing the constraints collected so far. A state $\langle \leftarrow a, B \,||\, C \rangle$ where a is selected literal can be *reduced* by one of the following rules:

1. if a is an atom $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n) \leftarrow B_1$ is a program clause then the new state will be
$$\langle \leftarrow s_1 = t_1, \ldots, s_n = t_n, B_1, B \,||\, C \rangle$$
2. if a is an equation $s = t$ where neither s nor t contain open functions and $\theta = mgu(s, t)$ then the new state is $\langle \leftarrow \theta(B) \,||\, \theta(C) \rangle$.
3. if a is a primitive constraint or equation containing an open function then the new state is $\langle \leftarrow B \,||\, a \wedge C \rangle$.
4. if a is a negative literal $not(p)$ and $\{\langle \Box \,||\, c_1 \rangle, \ldots, \langle \Box \,||\, c_n \rangle\}$ are the final states of all possible derivations starting from $\langle \leftarrow p \,||\, true \rangle$ then the new state will be $\langle \leftarrow \neg c_1, \ldots, \neg c_n, B \,||\, C \rangle$.

A *derivation* from a state S_0 is a sequence of states $S_0, S_1, \ldots, S_n, \ldots$ such that there is a reduction from S_{i-1} to S_i. A derivation from a goal G is a derivation from the state $\langle G \,||\, true \rangle$. A *successful* derivation is a *finite* derivation with a last state $\langle \Box \,||\, C \rangle$.

To use this procedure for solving the problems represented in our framework, we can write the integrity constraints as denials of the form $false \leftarrow L_1, \ldots, L_n$, add them to the program P and consider the goal $\leftarrow not(false)$. A successful derivation for this goal with a final state $\langle \Box \,||\, C \rangle$ will essentially evaluate the integrity constraints w.r.t. the program and reduce them to the formula C containing only primitive constraints. Then an interpretation of the open functions satisfying the answer constraint C can be extended to a model M of the program P such that $M \models IC$.

To obtain such a pre-interpretation we can give the formula C to a constraint solver. An important restriction for performing this step is that the arguments of all open functions appearing in the constraint formula C must be ground. Then in order to find an interpretation of the open functions we can replace every ground instance $f(d)$ of an open function f in C with a new variable $X_{f(d)}$ and obtain a new constraint formula C'. These new variables will stand for the interpretation of the particular ground instances of the open functions. The new formula C' can then be simplified to a conjunctive normal form and given to a finite domain constraint solver. If there is a solution then the bindings of the newly introduced variables will give the interpretation of the open functions.

7 Discussion

Declarative knowledge representation is a powerful paradigm. In the context of declarative specifications many computational problems are of abductive nature. However, abduction is computationally complex. Understanding all details of the inference rules requires a very good knowledge of logic. Moreover, it is very hard to understand how the formulation influences the performance of the execution. This contrasts with logic programming languages such as Prolog, where the programmer can exercise a rudimentary but comprehensive control by carefully ordering atoms in clauses.

In his paper we have started the exploration of logic programming extended with open functions. We have shown that finding an interpretation for the open function symbols which is consistent with the integrity constraints is a form of abduction. Throughout a number of examples, we have shown that it allows for elegant declarative formulations of prototypical problems in the areas of finite domain constraint programming and abductive programming. Also, the execution strategy becomes basically SLDNF, the only deviation is for unifications involving open function symbols, hence it should not be too hard for an experienced logic programmer to control the computational aspects.

In our proposed language, a program consists of:

- Declarations of open function symbols.
- Definitions of predicates as conventional logic programs.
- Integrity constraints, also written as logic programs, but allowing for clauses with disjunctions in the head.
- Declarations of the target predicates, the user defined predicates whose models make up the answer to the problem.

The kernel of the execution mechanism consists of evaluating the integrity constraints (after transforming them to denials) and to generate constraints on the open function symbols which ensure that the integrity constraints are satisfied by the solution. Contrary to full abductive systems, the evaluation stays close to conventional SLDNF evaluation. The main difference is that unifications have to be monitored to track the presence of open function symbols as their interpretation differs from the Herbrand interpretation and is unknown at evaluation time. The unifications involving open function symbols give rise to constraints which are handed over to a finite domain constraint solver which searches for an interpretation.

The use of an SLDNF like evaluation mechanism makes it feasible to incorporate a tabling mechanism as we have illustrated with the Hamiltonian path example.

We have built a very simple prototype as a meta-interpreter in Prolog. It is able to evaluate most of the examples described in the paper. However, much research remains to be done to reduce the overhead of monitoring all unifications for the presence of open function symbols. Techniques from abstract interpretation could be applied to analyze in which argument positions open function symbols can occur. Unifications not involving them could be directly executed

in the underlying Prolog system. This could drastically limit the overhead of monitoring unifications. Another source of overhead is that certain branches of the evaluation of integrity constraints can reach a state where the residual constraints involving the open function symbols turn out to be unsolvable, either by themselves or when combined with already derived constraints from other derivations. Carrying on such derivations is redundant, however detecting this would require a tight integration with the constraint solver and even then could be prohibitive expensive as search is inherent in verifying that a finite domain problem has a solution. Hence it may well be necessary to embed the use of open functions and of tabulation directly in a CLP language with facilities to control the processing of constraints.

References

1. M. Denecker. The well-founded semantics is the principle of inductive definition. In J. Dix, L. F. del Cerro, and U. Furbach, editors, *Logics in Artificial Intelligence, European Workshop, JELIA*, volume 1489 of *Lecture Notes in Computer Science*, pages 1–16, Dagstuhl, Germany, Oct. 1998. Springer.
2. M. Denecker and D. De Schreye. SLDNFA: an abductive procedure for abductive logic programs. *Journal of Logic Programming*, 34(2):111–167, Feb. 1998.
3. M. Denecker, H. Vandecasteele, D. De Schreye, G. Seghers, and T. Bayens. Scheduling by "abductive execution" of a classical logic specification. Compulog Network Meeting on Constraint Programming, Linz, 27-28, Oct. 1997.
4. H. Gao and D. S. Warren. A powerful evaluation strategy for CLP programs. In *Principles and Practice of Constraint Programming*, pages 90–97, Newport, Rode Island, 1993.
5. A. Kakas, R. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2(6):719–770, Dec. 1992.
6. A. C. Kakas and A. Michael. Integrating abductive and constraint logic programming. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 399–413. Tokyo, Japan, MIT Press, 1995.
7. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of 1994 ACM SIGMOD*. ACM Press, 1994.
8. H. Tamaki and T. Sato. OLD resolution with tabulation. In E. Shapiro, editor, *Third International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98, London, July 1986. Springer-Verlag.
9. B. Van Nuffelen and M. Denecker. Experiments for integration CLP and abduction. Draft, Dept. CS, K.U.LEUVEN, 1999.
10. D. S. Warren. Memoing for logic programs. *Commun. ACM*, 35(3):93–111, Mar. 1992.