User Guided Abductive Proof Generation for Answer Set Programming Queries

Avishkar Mahajan, Martin Strecker, Meng Weng Wong

User Guided Abductive Proof Generation for Answer Set Programming Queries

Avishkar Mahajan
Meng Weng Wong
Singapore Management University
Singapore

Martin Strecker
University of Toulouse
France

ABSTRACT

We present a method for generating possible proofs of a query with respect to a given Answer Set Programming (ASP) rule set using an abductive process where the space of abducibles is automatically constructed just from the input rules alone. Given a (possibly empty) set of user provided facts, our method infers any additional facts that may be needed for the entailment of a query and then outputs these extra facts, without the user needing to explicitly specify the space of all abducibles. We also present a method to generate a set of directed edges corresponding to the justification graph for the query. Furthermore, through different forms of implicit term substitution, our method can take user provided facts into account and suitably modify the abductive solutions. Past work on abduction has been primarily based on goal directed methods. However these methods can result in solvers that are not truly declarative. Much less work has been done on realizing abduction in a bottom up solver like the Clingo ASP solver. We describe novel ASP programs which can be run directly in Clingo to yield the abductive solutions and directed edge sets without needing to modify the underlying solving engine.

CCS CONCEPTS

• **Theory of computation → Constraint and logic programming**; **Automated reasoning**; • **Applied computing → Law**.

ACM Reference Format:
Avishkar Mahajan, Meng Weng Wong, and Martin Strecker. 2022. User Guided Abductive Proof Generation for Answer Set Programming Queries. In *24th International Symposium on Principles and Practice of Declarative Programming (PPDP 2022), September 20–22, 2022, Tbilisi, Georgia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3551357.3551383

1 INTRODUCTION

The goal of this paper is to show how a bottom up ASP reasoner like Clingo can be used for Abductive reasoning over First Order Horn clauses. As mentioned in the abstract previous work in abductive reasoning has mostly focused on implementing abduction in a top-down manner with Prolog as the underlying engine. CIFF [Mancarella et al. 2009] is a prominent example of this. More recently sCASP [Arias 2019; Arias et al. 2019] has been developed as a goal directed ASP implementation that can be used for abduction but this too uses a top down method for query evaluation. However there may be use cases where one wants to know all the resulting consequences of an abductive solution to a query with respect to a rule-set. Also, as mentioned in the abstract, top-down methods can sometimes result in solvers that are not truly declarative. Therefore an abductive reasoner that uses a solver like Clingo [Gebser et al. 2012] can complement the abilities of goal directed reasoners like sCASP, CIFF *etc.*

This paper shows how, given an input ASP rule set, one can write a new ASP program based on that rule set which will yield abductive solutions to queries, with the input ASP rule set as the background theory. The user does not have to explicitly specify the space of abducibles. This translation from the input ASP rule set to the derived ASP program is a purely mechanical one. The key idea is to encode backward chaining over the input rules through the use of meta predicates which incorporate a notion of 'reversing' the input rules to recursively generate pre-conditions from post conditions thereby generating a maximal space of abducibles. Then having generated this maximal space of abducibles, this 'feeds into' another part of the program where we have a representation of the input rules in the normal 'forward' direction. Entailment of the specified query is then checked via an integrity constraint and a minimal set of abduced facts is returned.

The main technical challenges are dealing with situations where input rules have existential variables in pre-conditions or when the query itself has existential variables. The other challenge is to control the depth of the abducibles generation process. The work that seems to come closest to ours is [Schüller 2016]. It too uses some similar meta predicates to encode backward chaining, and a forward representation of the rules to check for query entailment via integrity constraints.

However there are several novel features in our work. Firstly, depth control for abducible generation is done in a purely declarative way as part of the encoding itself without needing to call external functions or other pieces of software. Furthermore, adding facts to the program automatically gives an implicit form of term substitution where Skolem terms or other 'place-holder' terms occurring in abducibles are replaced away so that the resulting proof is simplified, without any need for an explicit representation of equality between terms. Past work on this topic such as [Schüller 2016] models equality between terms via an explicit equality predicate which may become unwieldy. Another approach to dealing with existential variables encountered during the abductive proof search is to simply ground all the rules over the entire domain of constants. However, this can often lead to too many choices for what an existential variable may be substituted for which may result in unexpected/unintuitive solutions. Our method avoids both of these techniques. We present three main sets of abductive proof

PPDP 2022, September 20–22, 2022, Tbilisi, Georgia

ACM ISBN 978-1-4503-9703-2/22/09…$15.00
https://doi.org/10.1145/3551357.3551383

generation encodings. One of the encodings only supports partial term substitution whereas the other two support full term substitution. Lastly, we also present an encoding which generates a set of directed edges representing a justification graph for the generated proof, where the graph can be of any desired depth.

The rest of the paper is organised as follows. First we give a brief introduction to Answer Set Programming and Abductive reasoning then, Section 2 defines the problem being tackled more formally. Section 3 presents the encodings that facilitate the abductive proof generation and directed edge generation. The sections that follow discuss some formal results regarding completeness, finiteness of abductive proof generation. We also discuss a formal result regarding term substitution. Finally Section 7 discusses future work and concludes.

An extended version of this paper [Mahajan et al. 2022] is available from *arXiv*, containing full proofs and further examples.

1.1 Answer Set Programming

Answer Set Programming (ASP) is a declarative language from the logic programming family. It is widely used and studied by knowledge representation and reasoning and symbolic AI researchers for its ability to model common sense reasoning, model combinatorial search problems etc. It incorporates the *negation-as-failure* operator as interpreted under the *stable model semantics*. Clingo is a well established implementation of ASP, incorporating additional features such as *choice rules* and optimization statements. We shall only briefly touch upon various aspects of ASP and Clingo here. The reader may consult [Gebser et al. 2012] for a more thorough description. Each rule in an ASP program consists of a set of body atoms. Some of these body atoms maybe negated via the negation as failure operator *not*. Rules with no pre-conditions are called facts. Given a set of rules R and a set of facts F, the Clingo solver computes all stable models of the ASP program $F \cup R$. For example given the fact $r(alpha)$ and the rules:

```
p(X):-r(X),not q(X).
q(X):-r(X), not p(X).
```

The solver will show us 2 models or answer sets given by $\{r(alpha), p(alpha)\}$ and $\{r(alpha), q(alpha)\}$. Note that as opposed to Prolog, Clingo is a bottom up solver meaning that it computes complete stable models (also known as answer sets) given any ASP program. An integrity constraint is formally speaking a rule whose post-condition is the boolean $false$. In ASP, integrity constraints are written as rules with no post-conditions and are used to eliminate some computed answer sets. For example given in the following ASP program

```
r(alpha).
p(X):-r(X),not q(X).
q(X):-r(X), not p(X).
:-q(X).
```

any answer set where some instantiation of q is true is eliminated. Hence we get just one answer set. $\{r(alpha), p(alpha)\}$.
We will now give a quick introduction to two features of Clingo that we will use throughout this paper. Namely *choice rules* and *weak constraints*. Weak constraints are also often known as optimization statements. Intuitively a choice rule is a rule where if the pre-conditions are satisfied then the post-condition may or may not be made true. The post-condition of a choice rule is enclosed in curly brackets. So given the following ASP program:

```
r(alpha).
{q(X)}:-r(X).
```

, where the rule is a choice rule the solver will give us 2 models namely $\{r(alpha)\}$, $\{r(alpha), q(alpha)\}$. If we modify the program by adding an integrity constraint like so:

```
r(alpha).
{q(X)}:-r(X).
:-q(X).
```

then we get just one model $\{r(alpha)\}$.
Weak constraints are used in Clingo to order answer sets by preference according to the atoms that appear in them. Without going into too much detail let us just explain the meaning of one kind of weak constraint which is the only kind that we will use in the paper namely:

```
:~a(X). [1@1,X]
```

Adding this to an ASP program, orders the answer sets of the program according to the number of distinct instantiations of the predicate a in the answer set. The answer set with the least number of instantiations of a is called the most *optimal* answer set.

1.2 Abductive Reasoning

Briefly, abduction is a reasoning process where given a background theory T, we wish to find a set of facts F such that $F \cup T$ is consistent and $F \cup T$ entails some goal g for some given entailment relation. Usually we also want F to be minimal in some well defined sense. Traditional Abductive Logic Programming has a long history, but we have our own definitions of what it means to formulate and solve an abductive reasoning problem and we will make all the relevant concepts/notions precise in the sections that follow.

2 ABDUCTIVE PROOF GENERATION TASK

2.1 Formal Setup

Definition 1 (Abductive Proof Generation Task). Given a source ASP rule set R, consider the tuple $\langle R, q, U, C, N \rangle$, which we will refer to as the *Abductive Proof Generation Task*. In this tuple, R denotes a set of input ASP rules, which we shall also refer to as the input rules or the source rules throughout the rest of this paper. q is either a possibly un-ground or partially ground positive atom or, a ground negation-as-failure atom. q intuitively represents the goal of our abductive reasoning process. In the context of an abductive proof generation task we may also sometimes refer to q as the *query*. The set U consists of 2 subsets, $U = U_f \cup U_a$. Here U_f is a set of user provided facts. U_a is a set of integrity constraints that prevents certain atoms from being abduced. Throughout the rest of this paper we may sometimes just refer to the set U as a whole making it clear what is contained in the subsets. C denotes a set of ASP constraints which constrain which atoms may or may not appear in

the complete model that results from the rules, user provided facts and abducibles. Finally we have the non-negative integer N. This acts as the depth control parameter for abductive proof generation.

Given an abductive proof generation task $\langle R, q, U, C, N\rangle$,let us define what we mean by a *General Solution* to the task

Definition 2 (General Solution). Given an abductive proof generation task $\langle R, q, U, C, N\rangle$, we say that F is a general solution to this task if:

(1) If q is a positive atom, it is in some answer set $\mathcal{A}$ of $F \cup U_f \cup R \cup C$ where the un-ground variables in q have been replaced with any set of ground terms. If q is a ground negation-as-failure (NAF) atom, then there exists an answer set $\mathcal{A}$ of $F \cup U_f \cup R \cup C$ such that q is *not* in $\mathcal{A}$.
(2) F does not violate any of the integrity constraints in U_a, ie. F does not contain abducibles that are specifically disallowed by the constraints in U_a.
(3) F does not have any atoms whose depth level is greater than N.

We will next state some assumptions we make on the set of input ASP rules R and then also define what we mean by the depth level of an atom. After this we shall exemplify all these definitions with an example.

2.2 Input ASP program

When considering any abductive proof generation task we make the following assumptions on the input ASP rule set R. We assume that each source ASP rule has exactly the following form:

```
pre_con_1(V1),pre_con_2(V2),...,pre_con_k(Vk),
    not pre_con_k+1(Vk+1),...,not pre_con_n(Vn) -> post_con(V).
```

We further make the folllowing assumptions:

(1) Each pre-condition $pre\_con_i(V_i)$ is atomic and so is the post-condition $post\_con(V)$.
(2) The *not* in front of the pre-conditions denotes *negation as failure* interpreted under the *stable model semantics*
(3) V_i is the set of variables occuring in the i^{th} pre-condition which is either $pre\_con_i(V_i)$ or *not* $pre\_con_i(V_i)$ and V is the set of variables occuring in the post condition $post\_con(V)$. We assume that $V \subseteq V_1 \cup V_2 \cup ... \cup V_n$.
(4) Each variable occurring in the post condition is universally quantified over, and each variable that occurs in some pre-condition but not the post condition is existentially quantified. In particular together with (3), this means that there are no existentially quantified variables in a rule post-condition.
(5) Each variable that occurs in a negation-as-failure pre-condition also occurs in some positive pre-condition.
(6) Each input rule of the form above is assigned some unique rule id.

We further assume that given any integrity constraint c in C, every variable in a negation-as-failure atom in c also occurs in some positive atom in c. We will now define the depth level of an atom with respect to a given input source ASP rule set R and query q.

Definition 3 (Depth level of an atom). Given a ASP rule set R and some ground or partially ground positive atom q which we shall call the query, we define a map $\phi_{R,q}$ that maps an arbitrary positive atom to a set of non-negative integers. We will describe this map rather informally. For any atom q' such that q' is obtained from q by replacing the variables in q, with some ground terms we have, $0 \in \phi_{R,q}(q')$. Now the rest of the definition is recursive. Given an atom a, the non-negative integer n, $n \in \phi_{R,q}(a)$ if and only if there exists a rule r in R and some substitution θ of the variables in r such that there exists some precondition (NAF or positive) p of r such that θ applied to p gives a and θ applied to the post condition of r gives some atom a' where $n - 1 \in \phi_{R,q}(a')$.

Given an atom a let $\phi^{min}_{R,q}(a)$ be -1 if the set $\phi_{R,q}(a)$ is empty and let $\phi^{min}_{R,q}(a)$ be the minimum member of the set $\phi_{R,q}(a)$ otherwise. Then $\phi^{min}_{R,q}(a)$ is defined to be the *depth level* of a with respect to the rule set R and query q. If q is a ground *NAF* atom then given some positive atom a $\phi^{min}_{R,q}(a)$ is simply given by $\phi^{min}_{R,\tilde{q}}(a)$, where $\tilde{q}$ is obtained from q by removing the *not* operator from in front of q.

For example if R consisted of the rules

```
a(X):-b(X).
b(X):-a(X).
```

and q was $a(X)$, meaning q is un-ground then we would have that given any term t, $\phi^{min}_{R,q}(a(t)) = 0$, $\phi^{min}_{R,q}(b(t)) = 1$, and for any other atom p, $\phi^{min}_{R,q}(p) = -1$.

3 DERIVED ASP PROGRAMS

3.1 A First Example

Before we give the details of the main sets of rule translations that allow abductive reasoning and justification generation, here is a simple example to illustrate some key ideas and what the desired output for an abductive problem is. Consider the rule set R given by the 3 rules

```
p(X,Y):-q(X,Y),s(Y).
p(X,Y):-g(X,Y).
d(X,Y):-g(X,Y).
```

Now let q be $p(john, james)$, let U consist only of a single constraint that disallows any instance of the predicate p from being abduced. Next let the set C contain a single constraint that disallows any instance of the predicate d, meaning that we require a stable model of the user given facts, the abducibles, and the rules, which does not contain any instance of the predicate d, finally let $N = 2$. Then for this problem the minimal abductive solution can be represented by *abducedFact(q(john,james)), abducedFact(s(james))*, which is what we want to get out of our encoding. Intuitively, the way we will solve this abductive reasoning problem is by first encoding the input rules such as the ones above in the usual forward direction. Then we will have a representation which corresponds to 'reversing' the rules, *i.e.* we go from post-conditions to pre-conditions. These 'reveresed' rules generate a maximal space of abducibles which then feed into the forward rule translation. Finally we will have integrity constraints that ensure that the atom which we want to be true (represented by q) is indeed entailed by the abductive solution. An adapted version of this 'reversed rule' representation also enables us to generate a set of directed edges corresponding to

a justification graph. The technical challenge in this process comes from finding a way to deal with existential variables and the depth of abducible generation.

3.2 Input Rule translations

3.2.1 Forward Translation. Given an input ASP rule

```
pre_con_1(V1),pre_con_2(V2),...,pre_con(Vk),
   not pre_con(Vk+1),...,not pre_con_n(Vn) -> post_con(V).
```

we translate it in the following way:

```
holds(post_con(V)):-holds(pre_con(V1)),...,holds(pre_con(Vk)), not
      holds(pre_con(Vk+1)), ..., not holds(pre_con(Vn)).
```

We repeat this for each source ASP rule. For each constraint in C, we simply enclose each atom in the constraint inside the *holds* predicate. For example if C contains the constraint : $-b(X, Y)$., (meaning that we require an abductive solution such that the there exists a stable model of the abduced facts, rules and user provided facts, which contains no instantiations of the predicate b), we encode that constraint as:

```
:-holds(b(X,Y)).
```

3.2.2 Generating Abducibles. Before diving into the details of the abducibles generation encoding let us give a brief intuition for some key meta-predicates and rules that will show up. Firstly the binary meta-predicate *query* has as its first argument an atom which may become a candidate for abduction and as its second argument an integer corresponding roughly to the depth level of that atom with respect to (R, q). Next the meta-predicate *explains* has as its first argument, an atom which forms a pre-condition of some input rule instantiation, and as its second argument the corresponding input rule instantiation post condition. The third argument of the *explains* meta predicate carries the depth of the atom in the first argument. The final key meta-predicate is *createSub*. *createSub* carries information about input rule instantiations. The first argument of *createSub* is a tuple which carries generated rule instantiations of a particular rule via the instantiations of the variables in the rule in some fixed order, the second argument of *createSub* is again an integer depth parameter. Let us now explain the general structure of some of the rules in the abducibles generation encoding to illustrate the purpose of these meta predicates. Firstly we have abduction generation rules with the structure:

```
createSub(...,N+1):-query(...,N),N<M,max_ab_lvl(M).
```

Intuitively in this rule, the first argument of the *query* meta-predicate generates an instantiation of an input rule where that atom is the post-condition of the rule. Skolem terms or other 'place-holder' terms are used for rules with existential variables in pre-conditions. Then we have abducible generation rules with the structure:

```
explains(...,N):-createSub(...,N).
```

Here a given *createSub* atom generates an *explains* atom where the first argument of the *explains* atom carries an input rule pre-condition given by the rule instantiation corresponding to the *createSub* atom and the second argument of the *explains* atom is the instantiation of the rule post-condition. Next we have abducible generation rules with the structure:

```
query(...):-explains(...).
```

Here the first two arguments of the *explains* meta predicate, get passed on to generate two instances of the *query* meta predicate. One can see from this that intuitively, a given *query* atom that carries an input rule post-condition generates a *createSub* atom that carries an input rule instantiation. This then generates an *explains* atom whose left hand side argument is a rule pre-condition which then generates a new *query* atom. This is the central part of the backward chaining process. The choice rule

```
{abducedFact(X)}:-query(X,N).
```

Then produces the abducibles. Next we will describe the general structure of two kinds of rules which are key to enable a notion of term substitution where user-input is taken into account to simplify the generated abductive proof. First we have

```
createSub(...):-createSub(...),holds(...).
```

and next we have

```
createSub(...):-createSub(...),query(...).
```

In the first kind of rule, arguments of *holds* atoms can 'combine' with instances of the *createSub* meta-predicate to yeild new instances of *createSub*. The intuition here is that if a certain instance of an input rule precondition/postcondition has been established via the *holds* predicate then this creates new substitutions for variables in that input rule which then leads to other input rule preconditions given by that substitution being included in the space of generated abducibles. The same intuition applies for abducible generation rules of the second kind, where instances of the *query* predicate create new input rule substitutions. It turns out that constructing these abducible generation rules in a 'naive' way can lead to infinite answer sets, when there are skolem terms involved, even when the integer depth argument of all these meta-predicates is bounded. Hence we have different encodings for when there are skolem terms involved versus when there no skolem terms involved. Let us now get into the technical details of how these rules are constructed. First we will need a way to assign appropriate skolem terms to existential variables in pre-conditions. Given some rule in our rule set, say rule r_j, We fix some order O_j on the variables occuring in the combined set of variables from the post and pre-conditions of the rule r_j. Now we will describe a skolemization map that assigns an existential variable in a pre-condition of r_j to a skolem term. Firstly, let the rule r_j carry unique integer id j. Let v be a variable that occurs in some rule precondition but not the post condition. Then under this skolemization map, the variable v gets mapped to

```
skolemFn_j_v_(V)
```

where V denotes the variables in the post-condition occuring in the order inherited from O_{r_j}. For example consider the rule r_1:

```
p(Y,X):- q(X),r(X,Y),s(Z).
```

Assume that this rule carries integer id 1. Let O_1 be $[X, Y, Z]$. Then the variable Z gets mapped to the skolem term $skolemFn\_1\_Z(X, Y)$.

3.2.3 AG1. Given an input ASP rule r in R

```
pre_con_1(V1),pre_con_2(V2),...,pre_con(Vk),not
      pre_con(Vk+1),...,not pre_con_n(Vn) -> post_con(V).
```

our first set of translated abducible generation rules $AG1$ is given by the following.

```
create_subs(sub_Inst_j((V_sk),N+1):-query(post_con(V),N),max_ab_lvl(M),
      N<M-1.
```

Here V_{sk}, denotes the ordered list O_j but with existential variables replaced by their skolem term counter parts. Here j is the integer id for the rule. The integer $M = N + 1$, where N is the fifth entry of the tuple representing the abduction task, which represents the maximum depth of an abducible. Next we have the following rules:

```
explains(pre_con_1(V1),post_con(V),N):-create_subs(sub_Inst_t((V),N).
explains(pre_con_2(V2),post_con(V),N):-create_subs(sub_Inst_t((V),N).
...
explains(pre_con_n(Vn),post_con(V),N):-create_subs(sub_Inst_t((V),N).
```

Here V denotes all the variables occuring in the rule in the order O_j.

3.2.4 AG2. Now we shall construct the second set of abducible generating rules $AG2$. Given O_j construct F_j by adjoining the character $V\_$ to each entry of O_j. So for our example above F_1 becomes $[V\_X, V\_Y, V\_Z]$. Given a pre-condition p occurring in rule r with id j, $M_{r,p}$ is an ordered list constructed as follows. The i_{th} element of $M_{r,p}$ is the i_{th} element of O_j if the i_{th} element of O_j is a variable which occurs in p. Otherwise, the i_{th} element of $M_{r,p}$ is given by the i_{th} element of F_r. Now for each negated or positive precondition p we have the following rule:

```
create_subs(sub_Inst_t(M_(r,p)),M-1):-create_subs(sub_Inst_t(F_r),N),
                    holds(p),max_ab_lvl(M), N<M.
```

Repeat this for each pre-condition. This is the set of rules $AG2$.

3.2.5 AG3. Finally $AG3$ consists of just the single rule:

```
query(X,N):-explains(X,Y,N),max_ab_lvl(M),N<M.
```

Let us consider another example. Consider the input ASP rule:

```
a(X):- b(X,Y,Z), not c(X), not d(Y) .
```

Say this rule r has rule id 5. Let O_5 be $[X, Y, Z]$. Here the encoding for the rule $AG1$ is:

```
create_subs(subs_Inst_5(X,skolemFn_5_Y(X),skolemFn_5_Z(X)),N+1):-
   query(a(X),N),max_ab_lvl(M),N<M-1.

explains(b(X,Y,Z),a(X),N):-create_subs(subs_Inst_5(X,Y,Z),N).

explains(c(X),a(X),N):-create_subs(subs_Inst_5(X,Y,Z),N).

explains(d(Y),a(X),N):-create_subs(subs_Inst_5(X,Y,Z),N).
```

AG2 is given by:

```
create_subs(subs_Inst_5(X,Y,Z),M-1):-
     create_subs(subs_Inst_5(V_X,V_Y,V_Z),N),
     holds(b(X,Y,Z)),max_ab_lvl(M),N<M.

create_subs(subs_Inst_5(X,V_Y,V_Z),M-1):-
create_subs(subs_Inst_5(V_X,V_Y,V_Z),N),
      holds(c(X)),max_ab_lvl(M),N<M.

create_subs(subs_Inst_5(V_X,Y,V_Z),M-1):-
create_subs(subs_Inst_5(V_X,V_Y,V_Z),N),
      holds(d(Y)),max_ab_lvl(M),N<M.
```

3.2.6 Supporting code for Abduction. Given the original problem $\langle R, q, U, C, N\rangle$, set $M = N + 1$. Then we have the following:

```
max_ab_lvl(M).
query(Q,0):-generate_proof(Q).
{abducedFact(X)}:-query(X,M).
holds(X):-abducedFact(X).

holds(X):-user_input(pos,X).
```

For any predicate p, say of arity n such that no instance of p may be abduced, we add the constraint.

```
:-abducedFact(p(X1,X2,...,Xn)).
```

If instead only a specific ground instance of p or a partially ground instance of p should be prevented from being abduced then we simply adapt the above constraint accordingly. For instance if p is a binary predicate and we want that no instance of p where the first argument is *alpha* should be abduced we have the constraint

```
:-abducedFact(p(alpha,X)).
```

Next we add the following *weak* constraint so that in the optimal abductive solution as few abducibles as possible are used.

```
:~abducedFact(X).[1@1,X]
```

3.2.7 Specifying the goal. Here is the code to encode the goal of the abductive reasoning process represented by the parameter q. If q is a ground atom say $p(a1, a2.., an)$ for some predicate p then we have:

```
generate_proof(p(a1,a2,...,an)).
goal:-holds(p(a1,a2,..,an)).
:- not goal.
```

Here the constraint

```
:- not goal.
```

ensures that the abduced facts together with the input rules actually do entail the goal. If on the other hand q is un-ground or only partially ground then we have the following. Say our goal is of the form $p(a, X, b, Y, Y)$, which means that X,Y are existential variables. Then for the example we have the following:

```
generate_proof(p(a,v1,b,v2,v2)).
goal:-holds(p(a,X,b,Y,Y)).
:- not goal.
```

Here $v1$, $v2$ are fresh constants. If q is a ground NAF atom say *not p* then we simply write

```
generate_proof(p).
goal:-not holds(p).
:- not goal.
```

Given $\langle R, q, U, C, N\rangle$ let the complete derived ASP program that uses $AG1$, $AG2$, $AG3$, the supporting code and the forward translation be called $P^{res}_{\langle R,q,U,C,N\rangle}$. We will now give a modified abduction generation encoding which can be used when no rule in R contains an existential variable. As mentioned before, it turns out that using this modified encoding on rules that have existential variables can lead to infinite answer sets. After giving this modified encoding we will explain in detail the encodings with the aid of an example.

3.3 Extending abduction generation space for rules without existential variables

When we have rules without existential variables, we can construct a larger space of abducibles without worrying about our ASP programs having infinite answer sets because there are now no skolem expressions. The encoding $AG1$ is the same as before but now clearly there will be no skolem terms. The new version of $AG2$ which we shall call $AG2_{exp}$ now becomes for each rule

```
create_subs(sub_Inst_t(M_(r,p)),N):-
   create_subs(sub_Inst_t(F_r),N),holds(p).
```

Notice that as opposed to the previous encoding, here the integer argument of the *createSub* predicate on the left hand side is N as opposed to $M-1$. Repeat this for each pre-condition p. Then for the post-condition of the rule p', we have:

```
create_subs(sub_Inst_t(M_(r,p')),N):-
   create_subs(sub_Inst_t(F_r),N),holds(p').
```

Here $M_{r,p'}$ is defined exactly the same way as $M_{r,p}$ for some pre-condition p. Next, for each rule and for each pre-condition p in the rule we have.

```
create_subs(sub_Inst_t(M_(r,p)),N):-
   create_subs(sub_Inst_t(F_r),N),query(p,L).
```

For the post-condition p' we have:

```
create_subs(sub_Inst_t(M_(r,p')),N):-
   create_subs(sub_Inst_t(F_r),N),query(p',L).
```

This completes the encoding $AG2_{exp}$. The adapted version of $AG3$, $AG3_{exp}$, is given by adding to $AG3$ one extra rule. So $AG3_{exp}$ is:

```
query(X,N):-explains(X,Y,N),max_ab_lvl(M),N<M.
query(Y,N-1):-explains(X,Y,N),max_ab_lvl(M),0<N,N<M.
```

Given $\langle R, q, U, C, N\rangle$ let the complete ASP program that uses $AG1_{exp}$, $AG2_{exp}$, $AG3_{exp}$, the supporting code and the forward translation be called $P^{exp}_{<R,q,U,C,N>}$

3.4 Discussion of Abduction space generation

3.4.1 Full term substitution. We first give an example of the expanded abduction space encoding to explain the intuition behind various parts of the encoding. Consider the rule set below that has no existential variables but which has negation as failure and where the goal is un-ground.

```
relA(X,Y):-relB(X,Y), relD(Y), not relE(Y).
relE(Y):-relD(Y), not relF(Y).
```

Let the goal q be $relA(P,R)$, where P,R are un-ground existential variables. Next suppose that the only constraint on abducibles is that no instantiation of $relA$ can be abduced and further suppose that the set of user provided facts is initially empty. Finally let $N = 4$. Here is the complete encoding for this problem.

```
max_ab_lvl(5).
% Encoding the goal
generate_proof(relA(v1,v2)).
query(X,0):-generate_proof(X).
goal:-holds(relA(P,R)).
:- not goal.

% forward translation
holds(relA(X,Y)) :- holds(relB(X, Y)),holds(relD(Y)), not
      holds(relE(Y)).
holds(relE(Y)) :- holds(relD(Y)), not holds(relF(Y)).

% AG1_exp
createSub(subInst_r1(X,Y),N+1) :-
      query(relA(X,Y),N),max_ab_lvl(M),N<M-1.
createSub(subInst_r2(Y),N+1) :- query(relE(Y)
      ,N),max_ab_lvl(M),N<M-1.

explains(relB(X, Y), relA(X,Y) ,N) :- createSub(subInst_r1(X,Y),N).
explains(relD(Y), relA(X,Y) ,N) :- createSub(subInst_r1(X,Y),N).
explains(relE(Y), relA(X,Y) ,N) :- createSub(subInst_r1(X,Y),N).


explains(relD(Y), relE(Y) ,N) :- createSub(subInst_r2(Y),N).
explains(relF(Y), relE(Y) ,N) :- createSub(subInst_r2(Y),N).


% AG2_exp for rule 1

createSub(subInst_r1(X,Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
      holds(relA(X,Y)).
createSub(subInst_r1(X,Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
      holds(relB(X,Y)).
createSub(subInst_r1(V_X,Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
      holds(relD(Y)).
createSub(subInst_r1(V_X,Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
      holds(relE(Y)).

createSub(subInst_r1(X,Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
      query(relA(X,Y),L).
createSub(subInst_r1(X,Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
      query(relB(X,Y),L).
createSub(subInst_r1(V_X,Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
      query(relD(Y),L).
createSub(subInst_r1(V_X,Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
      query(relE(Y),L).

% AG2_exp for rule 2

createSub(subInst_r2(Y),N) :- createSub(subInst_r2(V_Y),N),
      holds(relE(Y)).
createSub(subInst_r2(Y),N) :- createSub(subInst_r2(V_Y),N),
      holds(relD(Y)).
createSub(subInst_r2(Y),N) :- createSub(subInst_r2(V_Y),N),
      holds(relF(Y)).

```

```
44 createSub(subInst_r2(Y),N) :- createSub(subInst_r2(V_Y),N),
       query(relE(Y),L).
45 createSub(subInst_r2(Y),N) :- createSub(subInst_r2(V_Y),N),
       query(relD(Y),L).
46 createSub(subInst_r2(Y),N) :- createSub(subInst_r2(V_Y),N),
       query(relF(Y),L).
47
48 % AG3_exp
49 query(X,N):-explains(X,Y,N),max_ab_lvl(M),N<M.
50 query(Y,N-1):-explains(X,Y,N),max_ab_lvl(M),N<M,0<N.
51
52 % Supporting code
53 {abducedFact(X)}:-query(X,N).
54 holds(X):-abducedFact(X).
55 holds(X):-user_input(pos,X).
56
57
58 :~abducedFact(Y).[1@1,Y]
59 :-abducedFact(relA(X,Y)).
```

We will now discuss various parts of the encoding.

As mentioned earlier, the general idea is to recursively generate a maximal space of abducibles by 'reversing' the rules and then checking via the Forward Translation and encoding of the goal, which abducibles are needed for entailment of the original query. More specifically in line with the intuitive discussion from before, any atom of the form $query(h, i)$ generates an input rule instantiation where h is the post-condition that particular rule instantiation. Such rule instantiations are represented by the *createSub* atom. In the example above, this is done via lines 13, 14 of the encoding. Line 13 corresponds to instantiations of rule 1 and line 14 corresponds to instantiations of rule 2. Then any such *createSub* atom, generates the appropriate set of *explains* atoms. This is lines 16-18 for rule 1 in the example, and lines 21, 22 for rule 2. The first argument of an *explains* atom is a pre-condition or body atom corresponding to the rule instantiation given by the *createSub* atom. The second argument is the post-condition or head of the rule instantiation. We have one such *explains* atom for each rule pre-condition. Via line 49, the first argument of an *explains* atom becomes the first argument of a *query* atom. This new *query* atom then recursively generates more *query* atoms via the process described. Any *query* atom corresponds to a candidate for abduction via the choice rule in line 53. Any fact which is abduced must hold due to line 54. At this point, before moving ahead let us first briefly comment further upon the integer arguments occuring in the *explains*, *createSub* and *query* atoms.

The integer parameter roughly represents the depth of an abducible in the proof graph of the original query. When a *query* atom carrying the post-condition of a rule generates a rule instantiation like in line 13 for example, the integer argument of the corresponding *createSub* atom increases by one. Then an *explains* atom derived from the application of a rule like line 16 retains the same integer argument and so does the corresponding fresh *query* atom generated from the application of the rule on line 49. Note that a fresh *query* atom can only be created from an *explains* atom if the integer parameter of the *explains* atom is less than M. The use of these integer parameters is important when we need skolem functions/terms in our abducible generation encoding due to having rules with existentially quantified variables in pre-conditions. The use of these integer parameters allows us to control the depth of the abducible generating space thus preventing infinite answer sets even in the presence of skolem functions. We will discuss this more later on. For now let us turn our attention to some of the other parts of the encoding. The $AG2_{exp}$ encoding enables a notion of implicit term substitution in (minimal) abductive solutions. This set of rules creates new instantiations of the input rules based on which other atoms are true. As stated earlier, creating new instantiations of the core input rules via the *createSub* atoms, then allows new abducibles to be added to the generated space of abducibles. Let us illustrate some of these ideas with an example. Upon running the above ASP program as the optimal solution given by the solver is:

```
abducedFact(relD(v2))
abducedFact(relB(v1,v2))
abducedFact(relF(v2))
```

Now if $relB(john, james)$ is added to the set of user provided facts then firstly, due to line 55 $holds(relB(john, james))$ becomes true. Then we have the following instantiation of line 28.

```
createSub(subInst_r1(john,james),1):-
   createSub(subInst_r1(v1,v2),1),holds(relB(john,james)).
```

Hence due to lines 16 and 50, the atom $query(relA(john, james), 0)$ becomes true. This leads to the atoms $query(relD(james), 1)$ and $query(relF(james), 2)$ becoming true. Hence the atoms $relD(james)$ and $relF(james)$ become part of the space of abducibles and the solver gives us the new optimal solution:

```
abducedFact(relD(james))
abducedFact(relF(james))
```

On the other hand if we instead add the fact $relF(mary)$ to the initially empty set of user provided facts then we get the follwing instantiation of line 41:

```
createSub(subInst_r2(v2),2):-createSub(subInst_r2(v2),2),
holds(relF(mary)).
```

Hence the atom $createSub(subInst\_r2(mary), 2)$ becomes true. Via line 22, and line 50 the atom $query(relE(mary), 1)$ becomes true. We thus get the following instantiation of line 35:

```
createSub(subInst_r1(v1,mary),1):- createSub(subInst_r1(v1,v2),1),
      query(relE(mary),1).
```

Thus the atom $createSub(subInst\_r1(v1, mary), 1)$ becomes true, which then via say line 16 and line 50 causes the atom $query(relA(v1, mary), 0)$ to become true. Now, because of $query(relA(v1, mary), 0)$, $relD(mary)$, $relB(v1, mary)$ become part of the space of abducibles and the solver gives us the optimal abductive solution:

```
abducedFact(relD(mary))
abducedFact(relB(v1,mary))
```

and a similar result is obtained if we add an instance of the predicate $relD$ to the initially empty set of user provided facts. Thus with this encoding we have full implicit term substitution. The place holder or 'dummy' variables $v1$, $v2$, always get replaced away in the optimal abductive solution based on the user provided facts. A subtle point here is that there is no notion of equality between terms. We are not setting $v2 = mary$. We are instead enlarging the

space of abducibles in a systematic way based on user provided facts so that a more optimal solution which involves replacing the term $v2$ for the term *mary* can be realized. Note that adding an 'unrelated' fact such as say $relG(mary)$ will not enlarge the space of abducibles in any way. So in some sense what we have is a method to enlarge the space of abducibles in an 'economical' way while still supporting a notion of term substitution. We will formulate and prove a formal result regarding this notion of term substitution later on.

3.4.2 Partial term substitution. When skolem terms/function are used to handle existential variables, we have to use the non- expanded abducible generation encoding which forces us to give up on complete term substituion. This is because having the complete term substitution mechanism can result in programs that have infinitely large abducible spaces. To recover finiteness of the space of the abducibles we have to forgo full term substitution. What we get instead is a kind of partial term substitution mechanism where skolem terms may only sometimes be substituted for user provided terms. First let us examine why in the presence of skolem functions, even a subset of the expanded abduction generation encoding can lead to infinite answer sets.

Consider the rule set consisting of just the single rule

```
relA(X):-relB(X,Y),relA(Y).
```

Suppose the goal is $a(john)$ Consider the encoding below, which is a subset of the expanded abducible generation encoding.

```
1  max_ab_lvl(5).
2  query(relA(bob),0).
3  :-not holds(relA(bob)).
4
5  holds(relA(X)) :- holds(relB(X, Y)),holds(relA(Y)).
6
7  explains(relB(X, Y), relA(X) ,N) :- createSub(subInst_r1(X,Y),N).
8  explains(relA(Y), relA(X) ,N) :- createSub(subInst_r1(X,Y),N).
9
10
11 createSub(subInst_r1(X,skolemFn_r1_Y(X)),N+1) :- query(relA(X)
       ,N),max_ab_lvl(M),N<M-1.
12
13 createSub(subInst_r1(X,Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
       holds(relB(X, Y)).
14 createSub(subInst_r1(V_X,Y),N) :-
       createSub(subInst_r1(V_X,V_Y),N),holds(relA(Y)).
15
16
17 query(X,N):-explains(X,Y,N),max_ab_lvl(M),N<M.
18 {abducedFact(X)}:-query(X,N).
19 holds(X):-abducedFact(X).
20 holds(X):-user_input(pos,X).
21
22 :~abducedFact(Y).[1@1,Y]
23 :-abducedFact(relA(bob)).
```

Due to line 11 in the encoding we get the atom

```
createSub(subInst_r1(bob,skolemFn_r1_Y(bob)),1).
```

Then due to lines 7 and 8 of the encoding we get the atoms

```
query(relA(skolemFn_r1_Y(bob)),1)
      query(relB(bob,skolemFn_r1_Y(bob)),1).
```

Then via lines 11 and 7 and due to the atom

```
query(relA(skolemFn_r1_Y(bob)),1)
```

we get the atom

```
query(relB(skolemFn_r1_Y(bob), skolemFn_r1_Y(skolemFn_r1_Y(bob))),2)
```

Then due to lines 18, 19, we get the atom

```
holds(relB(skolemFn_r1_Y(bob), skolemFn_r1_Y(skolemFn_r1_Y(bob))
```

Then due to line 13 of the encoding and the atom

```
createSub(subInst_r1(bob,skolemFn_r1_Y(bob)),1)
```

we get the atom

```
createSub(subInst_r1(skolemFn_r1_Y(bob),
      skolemFn_r1_Y(skolemFn_r1_Y(bob))),1)
```

Then due line 8 and line 17, we get the atom

```
query(relA(skolemFn_r1_Y(skolemFn_r1_Y(bob)),1)
```

In this way we can see that with the encoding above we would have answer sets that contain atoms of the form $query(relA(skolemFn\_r1\_Y(\ldots), 1)$ for aribtrarily large skolem function nesting depth. Hence the encoding above leads to infinitely large answer sets.

Intuitively, the core problem is lines like 13, 14 where the skolem depth of terms in the *createSub* predicate has no relation with the integer argument of the *createSub* predicate, thus allowing for abducibles, where the skolem depth of the arguments inside predicates can be arbitrarily large despite having a finite maximum abduction depth level. The solution to this problem then is to replace the N occuring as the integer argument of the *createSub* predicate in the head of the rule on lines 13, 14 with $M - 1$, where the M corresponds to the argument of *max_ab_lvl*. This means that *query* atoms which occur due to the use of rules like line 13, 14 cannot further cause fresh *query* atoms to be added to the abducibles space via rules like the one on line 11.

As a result of this however we lose complete term substitution. Consider the following abduction problem. R is given by the following ASP rules:

```
relA(P):-relB(P,R),relD(R).
relB(P,R):-relA(R),relC(P).
```

Let q be the atom $relA(john)$, let U consist of the constraints : $-abducedFact(relA(john))$., : $-abducedFact(relB(X, Y))$. meaning that q cannot itself be abduced and no instantiation of the predicate $relB$ can be abduced. Let the set of user provided facts be empty for now. Let the set C also be empty and let $N = 4$. This is the non expanded abduction encoding for this problem.

```
1  max_ab_lvl(5).
2
3  % Encoding the goal
4  generate_proof(relA(john)).
5  goal:-holds(relA(john)).
6  :-not goal.
7  query(X,0):-generate_proof(X).
8
9  % Core rule translation
10 holds(relA(P)) :- holds(relB(P, R)),holds(relD(R)).
```

```
holds(relB(P, R)) :- holds(relA(R)), holds(relC(P)).

% AG1
createSub(subInst_r1(P,skolemFn_r1_R(P)),N+1) :- query(relA(P)
      ,N),max_ab_lvl(M),N<M-1.
createSub(subInst_r2(P,Q),N+1) :- query(relB(P, Q)
      ,N),max_ab_lvl(M),N<M-1.



explains(relB(P, R), relA(P) ,N) :- createSub(subInst_r1(P,R),N).
explains(relD(R), relA(P) ,N) :- createSub(subInst_r1(P,R),N).


explains(relA(R), relB(P,R) ,N) :- createSub(subInst_r2(P,R),N).
explains(relC(P), relB(P,R) ,N) :- createSub(subInst_r2(P,R),N).


% AG2 for rule 1
createSub(subInst_r1(P,R),M-1) :- createSub(subInst_r1(V_P,V_R),N),
      N<M, holds(relB(P, R)),max_ab_lvl(M).
createSub(subInst_r1(V_P,R),M-1) :-
      createSub(subInst_r1(V_P,V_R),N), N<M,
      holds(relD(R)),max_ab_lvl(M).

% AG2 for rule 2
createSub(subInst_r2(V_P,R),M-1) :-
      createSub(subInst_r2(V_P,V_R),N), N<M,
      holds(relA(R)),max_ab_lvl(M).
createSub(subInst_r2(P,V_R),M-1) :-
      createSub(subInst_r2(V_P,V_R),N), N<M,
      holds(relC(P)),max_ab_lvl(M).

% AG3
query(X,N):-explains(X,Y,N),max_ab_lvl(M),N<M.

% Supporting code
{abducedFact(X)}:-query(X,N).
holds(X):-abducedFact(X).
holds(X):-user_input(pos,X).


:~abducedFact(Y).[1@1,Y]
:-abducedFact(relA(john)).
:-abducedFact(relB(X,Y)).
```

Running this program in Clingo, we get the output

```
abducedFact(relC(john))
abducedFact(relD(skolemFn_r1_R(skolemFn_r1_R(john))))
abducedFact(relA(skolemFn_r1_R(skolemFn_r1_R(john))))
```

as the solution with the least number of abducibles. Now adding $relC(john)$ as a user provided fact gives the following smaller abductive solution

```
abducedFact(relD(skolemFn_r1_R(skolemFn_r1_R(john))))
abducedFact(relA(skolemFn_r1_R(skolemFn_r1_R(john))))
```

Now if we further add $relA(mary)$ to the set of user provided facts then we get as a minimal abductive solution the answer $relD(mary)$. This is because by after adding these facts, $holds(relB(john, mary))$ becomes true. Then by line 28 of the encoding $createSub(subInst\_r1(john, mary), 4)$ becomes true. Then by line 20, and line 36 $query(relD(mary), 4)$ becomes true which then gives us the minimal abductive solution. However if instead of adding the fact $relA(mary)$ we instead add the fact $relD(mary)$, then we do not get a substitution of terms and the minimal abductive solution is still

```
abducedFact(relD(skolemFn_r1_R(skolemFn_r1_R(john))))
abducedFact(relA(skolemFn_r1_R(skolemFn_r1_R(john))))
```

This is because by line 29, the atom $createSub(subInst\_r1(john, mary), 4)$ becomes true, which due to line 19 and line 36 makes $query(relB(john, mary), 4)$ true. However now this cannot cause the atom $query(relA(mary), 5)$ to become true because line 15 cannot apply due to the constraint on the integer argument of the *query* atom. So what we have can be regarded as a partial term substitution mechanism.

3.4.3 Replacing skolem functions by a single constant. Let us see how having term substitution as a derived effect via enlargement of the space of abducibles rather than doing term substitution through an explicit equality predicate allows us to better handle problems where the core rules have existential variables but we do not wish to use skolem functions in the abductive reasoning process. Recall that not having skolem functions allows us to get full term substitution without the possiblity of infinitely large answer sets. Consider the problem , $\langle R, q, U, C, N\rangle$ where R is the following input rule set:

```
relA(X):-relB(X,Y),relC(X,Y).
relB(X,Y):-relD(X,Y,Z),relE(X,Y,Z).
```

let our q be $relA(john)$. Let the initial set of user provided facts be empty, furthermore, suppose that no instance of $relA$ or $relB$ may be abduced. Finally let the set C be empty and let $N = 4$. Consider the following encoding

```
max_ab_lvl(5).
% Encoding the goal
generate_proof(relA(john)).
query(X,0):-generate_proof(X).
goal:-holds(relA(john)).
:- not goal.


% Core rule translation
holds(relA(X)) :- holds(relB(X, Y)),holds(relC(X,Y)).
holds(relB(X,Y)) :- holds(relD(X,Y,Z)), holds(relE(X,Y,Z)).

% AG1_exp
createSub(subInst_r1(X,extVar),N+1) :- query(relA(X)
      ,N),max_ab_lvl(M),N<M-1.
createSub(subInst_r2(X,Y,extVar),N+1) :- query(relB(X,Y)
      ,N),max_ab_lvl(M),N<M-1.

explains(relB(X, Y), relA(X) ,N) :- createSub(subInst_r1(X,Y),N).
explains(relC(X, Y), relA(X) ,N) :- createSub(subInst_r1(X,Y),N).

explains(relD(X,Y,Z), relB(X,Y) ,N) :-
      createSub(subInst_r2(X,Y,Z),N).
explains(relE(X,Y,Z), relB(X,Y) ,N) :-
      createSub(subInst_r2(X,Y,Z),N).


% AG2_exp for rule 1

createSub(subInst_r1(X,V_Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
      holds(relA(X)).
createSub(subInst_r1(X,Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
      holds(relB(X,Y)).
createSub(subInst_r1(X,Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
      holds(relC(X,Y)).

createSub(subInst_r1(X,V_Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
      query(relA(X),L).
createSub(subInst_r1(X,Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
      query(relB(X,Y),L).
```

```
createSub(subInst_r1(X,Y),N) :- createSub(subInst_r1(V_X,V_Y),N),
      query(relC(X,Y),L).


% AG2_exp for rule 2

createSub(subInst_r2(X,Y,V_Z),N) :-
      createSub(subInst_r2(V_X,V_Y,V_Z),N), holds(relB(X,Y)).
createSub(subInst_r2(X,Y,Z),N) :-
      createSub(subInst_r2(V_X,V_Y,V_Z),N), holds(relD(X,Y,Z)).
createSub(subInst_r2(X,Y,Z),N) :-
      createSub(subInst_r2(V_X,V_Y,V_Z),N), holds(relE(X,Y,Z)).


createSub(subInst_r2(X,Y,V_Z),N) :-
      createSub(subInst_r2(V_X,V_Y,V_Z),N), query(relB(X,Y),L).
createSub(subInst_r2(X,Y,Z),N) :-
      createSub(subInst_r2(V_X,V_Y,V_Z),N), query(relD(X,Y,Z),L).
createSub(subInst_r2(X,Y,Z),N) :-
      createSub(subInst_r2(V_X,V_Y,V_Z),N), query(relE(X,Y,Z),L).

% AG3_exp
query(X,N):-explains(X,Y,N),max_ab_lvl(M),N<M.
query(Y,N-1):-explains(X,Y,N),max_ab_lvl(M),N<M,0<N.

% Supporting code
{abducedFact(X)}:-query(X,N).
holds(X):-abducedFact(X).
holds(X):-user_input(pos,X).


:~abducedFact(Y).[1@1,Y]
:-abducedFact(relA(X)).
:-abducedFact(relB(X,Y)).
```

Note that in lines 15, 16 instead of using skolem functions we use a single fresh constant $extVar$ to represent the existential variable in both rules. Now, when we run the program we get the following optimal solution

```
abducedFact(relC(john,extVar))
abducedFact(relE(john,extVar,extVar))
abducedFact(relD(john,extVar,extVar))
```

Now because, term substitution is only a derived effect and there is no equality relation, it is possible for different instances of $extVar$ to get replaced (or not) by different constants upon the addition of some user provided facts. For instance upon adding the fact $relD(john, james, mary)$, we get the optimal solution:

```
abducedFact(relC(john,james))
abducedFact(relE(john,james,mary))
```

So some instances of $extVar$ from the original solution have been replaced by 'james' and others by 'mary'. What this means is that each occurence of $extVar$ in the original solution can be thought of as simply a place-holder for a term where each instance maybe a placeholder for a different term. When we use skolem functions instead this is simply more explicit because we have different skolem terms representing different existential variables. More formally in the first solution the variables $[X, Y, Z]$ get mapped to $[john, extVar, extVar]$ respectively. Upon the addition of the extra fact the we get the mapping $[X, Y, Z] \rightarrow [john, james, mary]$. Using an equality relation to get from the first solution to the second would be impossible because we would need both the following equalities to hold: $extVar = james$, $extVar = mary$. (Of course the above solution could be obtained if one simply grounds the rules over the entire domain of constants but as mentioned in the introduction, the methods in this paper are aimed at avoiding such a naïve grounding as in general, one may get too many substitutions for existential variables)

Given $< R, q, U, C, N >$, let this ASP program where we use $AG1_{exp}$, $AG2_{exp}$, $AG3_{exp}$ but replace all use of skolem terms with $extVar$ be called $P^{semi-res}_{<R,q,U,C,N>}$.
We will now turn to the problem of generating a set of directed edges corresponding the computed abductive solution.

3.5 Generating Justification Trees

Given a source rule

```
pre_con_1(V1),pre_con_2(V2),...,pre_con_k(Vk),not
      pre_con_k+1(Vk+1),...,not pre_con_n(Vn) ->
post_con(V).
```

For each positive pre-condition $pre\_con\_u(V_u)$, we add the following ASP rule:

```
causedBy(pos,pre_con_u(Vu), post_con(V),N+1):-holds(post_con(V)),
      holds(pre_con_1(V1)),
holds(pre_con_2(V2)),...,holds(pre_con_k(Vk)),not
      holds(pre_con_k+1(Vk+1)),...,
not holds(pre_con_n(Vn)),justify(post_con(V),N).
```

For each negative precondition $pre\_con\_f(V_f)$ we add the following ASP rule:

```
causedBy(neg,pre_con_f(Vf), post_con(V),N+1):-holds(post_con(V)),
      holds(pre_con_1(V1)),
holds(pre_con_2(V2)),...,holds(pre_con_k(Vk)),not
      holds(pre_con_k+1(Vk+1)),...,
not holds(pre_con_n(Vn)), justify(post_con(V),N).
```

3.5.1 *Supporting code for justification tree.*

```
justify(X,N):-causedBy(pos,X,Y,N), not user_input(pos,X),N<M,
      max_graph_lvl(M).
directedEdge(Sgn,X,Y):-causedBy(Sgn,X,Y,M).

justify(X,0):-gen_graph(X),not user_input(pos,X).

directedEdge(pos,userFact,X):-directedEdge(pos,X,Y),
      user_input(pos,X).

directedEdge(pos,userFact,X):-gen_graph(X), user_input(pos,X).
```

3.6 Discussion of Justification generation

The intuition for the justification graph encoding is that given some user provided facts F and an input rule set R, an atom a is only contained in a stable model M of F, R if either a is in F or there exists some rule r in R such that for some ground instantiation r_g of r, all the pre-conditions of r_g, (ie. the body atoms) are true in M and the post-condition (ie. head) of r_g is a. Here the truth value of NAF atoms is interpreted in the usual way. The edges for the justification graph are calculated recursively. An atom $justify(h, k)$ represents the fact that $holds(h)$ needs to be justified. If r_g is a ground instantiantion of an input rule where the post condition of r_g is h and all the pre-conditions of r_g are true then for every positive precondition b_i of r_g we have the atom $causedBy(pos, b_i, h, k + 1)$ and for every

NAF pre-condition b_j we have the atom $causedBy(neg, b_j, h, k+1)$. Then if $k < M$, where we have $max\_graph\_lvl(M)$ for some integer value of M, we get the atoms $justify(b_i, k+1)$, for every positive pre-condition b_i which is not a user provided fact. Finally each *causedBy* atom generates a *directedEdge* atom, and these atoms are the set of directed edges representing the justification graph.

3.7 Some example executions

Given the following program

```
gen_graph(relA(john)).
max_graph_lvl(5).

user_input(pos,relE(john,james,mary)).
user_input(pos,relD(john,james,mary)).

holds(X):-user_input(pos,X).

holds(relA(X)) :- holds(relB(X, Y)),not holds(relC(X,Y)).
holds(relB(X,Y)) :- holds(relD(X,Y,Z)), holds(relE(X,Y,Z)).

causedBy(pos,relB(X,Y),relA(X),N+1):-holds(relA(X)),holds(relB(X,
      Y)),not holds(relC(X,Y)),justify(relA(X),N).
causedBy(neg,relC(X,Y),relA(X),N+1):-holds(relA(X)),holds(relB(X,
      Y)),not holds(relC(X,Y)),justify(relA(X),N).

causedBy(pos,relD(X,Y,Z),relB(X,Y),N+1):-holds(relB(X,Y)),
holds(relD(X,Y,Z)),holds(relE(X,Y,Z)),justify(relB(X,Y),N).
causedBy(pos,relE(X,Y,Z),relB(X,Y),N+1):-holds(relB(X,Y)),
holds(relD(X,Y,Z)),holds(relE(X,Y,Z)),justify(relB(X,Y),N).

justify(X,N):-causedBy(pos,X,Y,N), not user_input(pos,X),N<M,
      max_graph_lvl(M).
directedEdge(Sgn,X,Y):-causedBy(Sgn,X,Y,M).

justify(X,0):-gen_graph(X),not user_input(pos,X).

directedEdge(pos,userFact,X):-directedEdge(pos,X,Y),
      user_input(pos,X).

directedEdge(pos,userFact,X):-gen_graph(X), user_input(pos,X).
```

We get the following set of directed edges representing the justification graph.

```
directedEdge(pos,relB(john,james),relA(john))
directedEdge(pos,relE(john,james,mary),relB(john,james))
directedEdge(pos,relD(john,james,mary),relB(john,james))
directedEdge(neg,relC(john,james),relA(john))
directedEdge(pos,userFact,relD(john,james,mary))
directedEdge(pos,userFact,relE(john,james,mary))
```

4 SIMPLE ABDUCTIVE PROOF GENERATION TASK

We shall define here the notion of a *Simple Abductive Proof Generation Task*, as all of our formal results will apply to this restricted class of abductive proof generation tasks.

Definition 4 (Simple Abductive Proof Generation Task). Given an abductive proof generation task $\langle R, q, U, C, N\rangle$, we say that this task is a *simple abductive proof generation task* if the following hold:

(1) R contains no negation as failure.
(2) R contains no function symbols, arithmetic operators.
(3) No post condition of any rule in R contains repeated variables. For example the rule: $p(X, X) : -r(X, X, Y)$ is not allowed but the rule: $p(X, Y) : -r(X, X, Y)$ is allowed.
(4) C is empty.
(5) Any constraint on abducibles in U_a must consist of only a single positive fully un-ground atom with no repeated variables amongst its arguments. For example if p is a binary predicate, then the constraint : $-abducedFact(p(X, Y))$. is allowed but the constraint : $-abducedFact(p(X, X))$. Constraints where more than one atom appears are also not allowed. For instance the following would be disallowed: : $-abducedFact(p(X, Y)), abducedFact(r(X))$. Finally, constraints containing partially or fully ground atoms are disallowed. For example the following would be disallowed : $-abducedFact(p(james, Y))$.
(6) If U_a is such that no instance of some predicate p can be abduced, then U_f must not contain any instantiation of p.
(7) q must be positive and fully ground.

5 FINITENESS AND COMPLETENESS PROPERTIES OF SIMPLE TASKS

Theorem 5 (Finiteness). *Assume that $\langle R, q, U, \emptyset, N\rangle$ is such that it is a simple abductive proof generation task. Then $P^{res}_{\langle R,q,U,\emptyset,N\rangle}$ cannot have infinite answer sets.*

The proof can be found in [Mahajan et al. 2022].

Theorem 6 (Completeness). *Given a simple abductive proof generation task, if there exists a general solution S to that task then there exists a ASP solution S_{ASP} to that task corresponding to an answer set of the ASP program $P^{res}_{\langle R,q,U,\emptyset,N\rangle}$.*

The proof can be found in [Mahajan et al. 2022].

Let us just comment on the results above in a slightly broader context . Firstly it is not difficult to see that the above results for finiteness and completeness hold for a slightly larger class of abductive proof generation tasks than the class of simple tasks. Namely we can in fact relax condition 7 in the definition of simple tasks, to allow q to be un-ground or only partially ground. Call this class of abduction tasks *semi-simple*. Also, the completeness result in fact holds if $P^{res}_{\langle R,q,U,\emptyset,N\rangle}$ is replaced with $P^{semi-res}_{\langle R,q,U,\emptyset,N\rangle}$. In summary what we have then is the following
Given a *semi-simple* task $\langle R, q, U, \emptyset, N\rangle$. $P^{res}_{\langle R,q,U,\emptyset,N\rangle}$, $P^{semi-res}_{\langle R,q,U,\emptyset,N\rangle}$, both enjoy the completeness and finiteness properties. However only $P^{semi-res}_{\langle R,q,U,\emptyset,N\rangle}$ supports full implicit term substitution whereas $P^{res}_{\langle R,q,U,\emptyset,N\rangle}$ only supports partial term substitution. We shall formulate and prove a formal result regarding term substitution for $P^{semi-res}_{\langle R,q,U,\emptyset,N\rangle}$ next.

6 PROOF SIMPLIFICATION USING USER PROVIDED FACTS

Definition 7 (Abstract Proof Graph). Given a rule set R which does not contain NAF, a predicate p and integer n define the abstract proof graph $G_{R,p,n}$ as follows. The nodes of $G_{R,p,n}$ is the set of *query* predicates generated by the rules just by the rules $AG1$ and $AG3$, in the encoding $P^{res}_{\langle R,q,U,C,N\rangle}$ where in $\langle R, q, U, C, N\rangle$, q is $p(v1, v2.., vk)$ assuming p has arity k, U, C are empty and $N = n$. The edge relation is defined as follows. Two nodes $d1$, $d2$ are

connected by a directed edge represented as $E(d1, d2)$ if and only if, $d1$ represents a pre-condition of an input rule where $d2$ is the post condition.

So if R' consisted of the rules:

```
a(X):-b(X,Y),c(Y).
b(X,Y):-d(X,Y,Z).
```

Then $G_{R',a,2}$ is:

```
E(query(b(v1,sk(v1)),1),query(a(v1),0)),
E(query(c(sk(v1)),1),query(a(v1),0)))
E(query(d(v1,sk(v1),sk'(v1,sk(v1))),2),query(b(v1,sk(v1)),1))
```

Here sk sk' are just abbreviations of the full skolem function names. Also we assume the order $[X, Y, Z]$ on variables in the second rule, and the order $[X, Y]$ on variables in the first rule. (Recall that when defining the abducible generation rules in section 3 we had an order on variables in a rule)

Definition 8 (Abstract Instance Set). Given a rule set R, predicate p and integer n, the corresponding Abstract Instance Set denoted corresponding to this triple denoted by $I_{R,p,n}$ is the set of *create_Sub* predicates generated by the rules just by the rules $AG1$ and $AG3$, in the encoding $P^{res}_{\langle R,q,U,C,N\rangle}$ where in $\langle R, q, U, C, N\rangle$, q is $p(v1, v2.., vk)$ assuming p has arity k, U, C are empty and $N = n$.

So for the example above the set $I_{R',a,2}$ is:
$createSub(subInst\_r1(v1, sk(v1)), 1)$,
$createSub(subInst\_r2(v1, sk(v1), sk'(v1, sk(v1))), 2)$

Definition 9 (Minimal abstract Proof Graph). Given an abstract proof graph $G_{R',p,N}$, construct the minimal proof graph $G^{min}_{R',p,N}$ as follows. Firstly for given an integer k, going from left to right, delete all duplicate nodes $query(a', k)$ such that $query(a, k)$ is already in the proof graph and $a' = a$. Next for going down the proof graph, for each $k \in 0, 1..., N$, delete the node $query(a, k)$, if there exists $l < k$, such that $query(a', l)$ is in the proof graph $a = a'$. This forms $G^{min}_{R',p,N}$. The edge relation is inherited from $G_{R,p,N}$ in the obvious way.

Note firstly that if for some b, j, $query(b, j)$ is in $G_{R',p,N}$, then there exists some $h \leq j$ such that $query(b, h)$ is in $G^{min}_{R',p,N}$. Secondly we have the following property:

Lemma 1. *Given $query(a, k)$ in $G^{min}_{R',p,N}$, unless $k = 0$, there exists $query(a', k-1)$ in $G^{min}_{R',p,N}$ such that we have*
$E(query(a, k), query(a', k-1))$.

Proof. *(sketch).*
Let $query(a'', k-1)$ be such that $E(query(a, k), query(a'', k-1))$ is an edge of $G_{R',p,N}$. Now if $query(a'', k-1)$ is not in $G^{min}_{R',p,N}$, then there exists $s < k-1$ such that $query(a'', s)$ is in $G^{min}_{R',p,N}$. Therefore $query(a'', s)$ is in $G_{R',p,N}$. Then it follows that $query(a, s+1)$ is in $G_{R',p,N}$. However this is a contradiction since $s + 1 < k$. Hence it must be the case that $query(a'', k-1)$ is in $G^{min}_{R',p,N}$. □

For the example we have been considering, the minimal abstract proof graph is the same as the abstract proof graph.

Definition 10 (Concrete Proof Graph). Given a minimal abstract proof graph $G^{min}_{R,p,n}$, and a substitution θ for terms in the minimal abstract proof graph, define the concrete proof graph $C_{R,p,n,\theta}$ to be the set of *query* atoms obtained after doing the substitution θ on the set of *query* atoms in $G^{min}_{R,p,n}$. (We drop the *min* from the notation for the concrete proof graph as we will always mean a substitution from the minimal abstract proof graph)

Note that such a substitution θ need not be injective.

Definition 11 (Parent node, child node, sibling node, descendant). Given any concrete or abstract, minimal or non minimal proof graph G and given two nodes $d1, d2$ in G, we say that $d2$ is a *parent* of $d1$ if we have $E(d1, d2)$. In such a case we say $d1$ is a *child* of $d2$. Given two nodes $d3$ and $d4$, we say $d3$ is a *sibling* node of $d4$, if there exists some node d' such that $E(d3, d')$ and $E(d4, d')$. ($d4$ is then also a sibling node of $d3$) Given two nodes $d5$ and $d6$, we say $d5$ is a *descendant* of $d6$ if $E_{tr}(d5, d6)$ is true where E_{tr} is the transitive closure of E.

Definition 12 (Concrete Instance Set). Given an abstract instance set $I_{R,p,n}$, and a substitution θ for terms in the instance set, define the concrete instance set $I_{R,p,n,\theta}$ to be the set of *createSub* atoms obtained after applying the substitution θ on the set of *createSub* atoms in $I_{R,p,n}$, (which itself is associated to the full abstract proof graph rather than the minimal one).

For our example consider the substitution $\theta = [v1 \rightarrow john, sk(v1) \rightarrow extVar, sk'(v1, sk(v1)) \rightarrow extVar]$. Then $C_{R',a,2,\theta}$ is :

```
E(query(b(john,extVar),1),query(a(john),0)),
E(query(c(extVar),1),query(a(john),0)))
E(query(d(john,extVar,extVar),2),query(b(john,extVar),1))
```

$I_{R,p,n,\theta} = createSub(subInst\_r1(john, extVar), 1)$,
$createSub(subInst\_r2(john, extVar, extVar), 2)$

Definition 13 (Derived Substitution - T). Given some $G^{min}_{R,p,n}$, and an associated $C_{R,p,n,\theta}$, let q_c be a *query* atom from $C_{R,p,n,\theta}$. Let S_{q_c} be the set of *query* atoms in $G^{min}_{R,p,n}$, which upon application of the substitution θ give q_c. Now pick an atom q_o from S_{q_c}. Now suppose q_c is given by $query((t_1, t_2, ..., t_j), k)$, q_o is given by $query((e_1, e_2, ..., e_j), k)$. Now consider an arbitrary query atom q_f given by $query((a_1, a_2, ..., a_j), k)$, which is such that the map ψ mapping each e_i to the corresponding a_i is well defined. (ie. ψ is not one-to-many). Then define the substituion $\phi = T(\theta, q_c, q_o, q_f)$ on terms in $G^{min}_{R,p,n}$ by the following:
For a term u in $G^{min}_{R,p,n}$ (meaning u occurs as the argument of the predicate inside some *query* atom), if u is in the set $\{e_1, e_2, ..., e_j\}$ then $\phi(u) = \psi(u)$, otherwise $\phi(u) = \theta(u)$.

For instance going back to our example if we let q_c be $query(b(john, extVar), 1)$ let q_o be $query(b(v1, sk(v1)), 1)$ and let q_f be $query(b(john, james), 1)$ then $\phi = T(\theta, q_c, q_o, q_f)$ is the substitution $\theta = [v1 \rightarrow john, sk(v1) \rightarrow james, sk'(v1, sk(v1)) \rightarrow extVar]$. We now have the following theorem:

Theorem 14 (Term substitution). *Consider the abductive proof generation task $\langle R, q, U, C, N\rangle$, and suppose that his task is a simple abductive proof generation task. Let p be the predicate corresponding to q. Suppose A is an answer set of $P^{semi-res}_{<R,q,U,C,N>}$ and let A contain*

$C_{R,p,N,\theta}$ and $I_{R,p,N,\theta}$. Now say the atom q_c $query(p_i(t_1, t_2, .., t_j), k)$ is in $C_{R,p,N,\theta}$. Let q_o be some query atom from the set S_{q_c} given by $query((e_1, e_2, ..., e_j), k)$. Now suppose, $q_f = query((a_1, a_2, ..., a_j), k)$ is an arbitrary query atom such that the map ψ from the e_is to the a_is as described in the definition above is well defined. Then upon adding the fact $query(p_i(a_1, a_2, .., a_j), k)$ to $P^{semi-res}_{<R,q,U,C,N>}$, the resulting program has an answer set A' such that A' contains $C_{R,p,N,\phi}$ and $I_{R,p,N,\phi}$ where $\phi = T(\theta, q_c, q_o, q_f)$

The proof can be found in [Mahajan et al. 2022].

Corollary 1 (Adding facts). *Given the simple abductive proof generation task $\langle R, q, U, C, N\rangle$, let p be the predicate corresponding to q. Suppose A is an answer set of $P^{semi-res}_{<R,q,U,C,N>}$ and let A contain $C_{R,p,N,\theta}$ and $I_{R,p,N,\theta}$. Now say the atom q_c $query(p_i(t_1, t_2, .., t_j), k)$ is in $C_{R,p,N,\theta}$. Let q_o be some query atom from the set S_{q_c} (The set of pre-images of q_c in $G^{min}_{R,p,N}$) given by $query((e_1, e_2, ..., e_j), k)$. Now suppose, $h_f = holds((a_1, a_2, ..., a_j))$ is an arbitrary holds atom such that the map ψ from the e_is to the a_is as described in the earlier definition of derived substitution is well defined. Then upon adding the fact $holds(p_i(a_1, a_2, .., a_j))$ to $P^{semi-res}_{<R,q,U,C,N>}$, the resulting program has an answer set A' such that A' contains $C_{R,p,N,\phi}$ and $I_{R,p,N,\phi}$, where $\phi = T(\theta, q_c, q_o, q_f)$, where $q_f = query(p_i(a_1, a_2, .., a_j), k)$.*

The proof can be found in [Mahajan et al. 2022].

The preceding theorem and corrolary correspond to the notion of full implicit term substitution which we discussed earlier. For example going to our main example in this section. Let R be the set of rules:

```
a(X):-b(X,Y),c(Y).
b(X,Y):-d(X,Y,Z).
```

Let q be $a(john)$, let the set of user provided facts be empty and suppose that no instances of the predicate a or b can be abduced. Finally suppose that the set C is empty and $N = 2$. Then upon running the ASP program $P^{semi-res}_{\langle R,q,U,C,N\rangle}$ we will get the minimal abductive solution

```
abducedFact(c(extVar)), abducedFact(d(john,extVar,extVar))
```

This corresponds to the substitution $\theta = [v1 \rightarrow john, sk(v1) \rightarrow extVar, sk'(v1, sk(v1)) \rightarrow extVar]$. Then upon modifying the set of user provided facts by adding the fact $c(james)$, we get the smaller abductive solution:

```
abducedFact(d(john,james,extVar))
```

which corresponds to the substitution $\theta = [v1 \rightarrow john, sk(v1) \rightarrow james, sk'(v1, sk(v1)) \rightarrow extVar]$.

As was the case for the finiteness and completeness results, it is in fact the case the theorem and corollary proved in the previous section hold for the slightly larger class of abductive proof generation tasks which we called semi-simple. Overall, we have the following results. Given a semi-simple task $\langle R, q, U, \emptyset, N\rangle$, if no rule in R contains existential variables, then to solve this task we can use the encoding $P^{exp}_{\langle R,q,U,\emptyset,N\rangle}$, which enjoys the completeness, finiteness and full term substitution properties. If R does contain existential variables then we can either use the encoding $P^{res}_{\langle R,q,U,\emptyset,N\rangle}$ which enjoys the properties of completeness and finiteness but only gives us partial term substitution or we can use $P^{semi-res}_{\langle R,q,U,\emptyset,N\rangle}$ which enjoys the properties of completeness, finiteness and full term substitution.

7 CONCLUSIONS

We have presented several encodings for abductive proof generation in ASP, incorporating notions of depth control and novel implementations of term substitution. We have also given an encoding that allows one to generate a set of directed edges representing a justification graph.

It seems to us that some of the ideas involved in the term substituion mechanism are similar to the ideas involved when one uses *Sideways Information Passing Strategies* [Beeri and Ramakrishnan 1991] to re-write datalog rules for more efficient evaluation of queries by incorporating elements of top-down reasoning. However we have not explored this connection in detail. Those techniques typically involve a complete re-write of the input rules according some chosen fixed sideways information passing strategy, which makes that whole approach quite different to ours. [Stickel 1994] describes an approach to doing abductive reasoning in a bottom up manner. He uses 'continuation predicates' to pass substitutions from previously evaluated rule pre-conditions to rule pre-conditions yet to be evaluated, given the rule post-condition as the 'goal'. This somewhat resembles our use of 'createSub' predicates. However it seems to us that that approach imposes a strict order on the evaluation of preconditions of a rule, which makes that method much less general than ours.

There are several possible directions for future work. One possible line of theoretical investigation could be to study how the abductive solutions calculated by our methods (and any resulting extra consequences of the abductive solution and input rules) could be generalised to sentences in first order logic. Roughly speaking, given an abductive solution involving instances of 'extVar' the aim would be to map these solutions to solutions where instances of 'extVar' are replaced by universally quantified variables (where perhaps such a variable may not take values from some finite set). Distinguishing between instances of 'extVar' that should get mapped to distinct universally quantified variables can be done by adding certain facts that would result in the generated abductive solution being modified so that the 'matching' occurrences of 'extVar' get replaced by some other fresh constant. Intuitively, it seems to us that our method of calculating and simplifying abductive solutions without grounding over the entire domain of constants gives an appropriate setting to explore some of these ideas. Of course the correctness/applicability of such techniques would have to be investigated in a rigorous and formal manner.

Another possible future line of work may include extending the formal results presented here to a larger class of abductive proof generation problems. It also seems to us that the main technique used to generate the directed edge set representing the justification graph could be adapted for use in SAT/SMT solvers to get justifications out of them.

We also have yet to study the complexity problems associated with the methods presented in this paper. [Eiter et al. 1997] provides a thorough study of the complexity of abductive reasoning. It remains to be seen how those results, many of which deal with

propositional logic, could be carried over to our setting, where we aim to compute abductive solutions without complete grounding of rules.

ACKNOWLEDGMENTS

This research is supported by the National Research Foundation, Singapore, under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

Joaqu'in Arias. 2019. *Advanced Evaluation Techniques for (Non)-Monotonic Reasoning Using Rules with Constraints.* Ph.D. Dissertation. Universidad Politécnica de Madrid.

Joaquín Arias, Manuel Carro, Zhuo Chen, and Gopal Gupta. 2019. Constraint Answer Set Programming without Grounding and its Applications. In *Datalog 2.0 (CEUR Workshop Proceedings, Vol. 2368)*, Mario Alviano and Andreas Pieris (Eds.). CEUR-WS.org, Philadelphia, PA (USA), 22–26. http://ceur-ws.org/Vol-2368/paper2.pdf

Catriel Beeri and Raghu Ramakrishnan. 1991. On the power of magic. *The Journal of Logic Programming* 10, 3 (1991), 255–299. https://doi.org/10.1016/0743-1066(91)90038-Q Special Issue: Database Logic Progamming.

Thomas Eiter, Georg Gottlob, and Nicola Leone. 1997. Abduction from Logic Programs: Semantics and Complexity. *Theor. Comput. Sci.* 189, 1-2 (1997), 129–177. https://doi.org/10.1016/S0304-3975(96)00179-X

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2012. *Answer Set Solving in Practice.* Morgan & Claypool Publishers, Uni Potsdam.

Avishkar Mahajan, Martin Strecker, and Meng Weng Wong. 2022. User Guided Abductive Proof Generation for Answer Set Programming Queries (Extended version). Forthcoming. https://arxiv.org/

Paolo Mancarella, Giacomo Terreni, Fariba Sadri, Francesca Toni, and Ulle Endriss. 2009. The CIFF proof procedure for abductive logic programming with constraints: Theory, implementation and experiments. *Theory Pract. Log. Program.* 9, 6 (2009), 691–750. https://doi.org/10.1017/S1471068409990093

Peter Schüller. 2016. Modeling Variations of First-Order Horn Abduction in Answer Set Programming. *Fundam. Informaticae* 149, 1-2 (2016), 159–207. https://doi.org/10.3233/FI-2016-1446

Mark E. Stickel. 1994. Upside-Down Meta-Interpretation of the Model Elimination Theorem-Proving Procedure for Deduction and Abduction. *J. Autom. Reason.* 13, 2 (1994), 189–210. https://doi.org/10.1007/BF00881955