Probabilistic Abductive Logic Programming in Constraint Handling Rules

Henning Christiansen

Research group PLIS: Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: henning@ruc.dk

Abstract. A class of Probabilistic Abductive Logic Programs (PALPs) is introduced and an implementation is developed in CHR for solving abductive problems, providing minimal explanations with their probabilities. Both all-explanations and most-probable-explanations versions are given. Compared with other probabilistic versions of abductive logic programming, the approach is characterized by higher generality and a flexible and adaptable architecture which incorporates integrity constraints and interaction with external constraint solvers. A PALP is translated in a systematic way into a CHR program which serves as a query interpreter, and the resulting CHR code describes in a highly concise way, the strategies applied in the search for explanations.

1 Introduction

Logic programs provide a very flexible and general representation scheme for knowledge about complex and interrelated phenomena. Deductive reasoning, i.e., reasoning about what is known, in logic programs may be done within the Prolog programming language, and various extensions for synthetic reasoning such as abduction and induction has been developed. Abductive reasoning means to search for missing world facts, or we may say values of hidden variables, which can explain observations of the state of affairs. Applications for diagnosis in medicine and fault finding in mechanical or virtual systems, are some of the obvious applications.

Abduction has been approached using Constraint Handling Rules, CHR, [13] initially by [1] and later extended and applied in different context by [7, 5, 4, 8]. See [16] for an overview paper on this direction.

In the present paper, we investigate how abduction extended with probabilities can be implemented in CHR. We suggest a method of compiling a class of Probabilistic Abductive Logic Programs into CHR programs which serve as query interpreters. For ease of understanding, we show first an all-solutions version for propositional programs and the extend to the general case and sketch also a best-first version.

The present paper is an extract of a longer article [17] which includes also proofs of central properties; the present approach is generalized to a larger class of Prioritized Abductive Logic Programs in [18].

2 Probabilistic Abductive Logic Programs

2.1 Syntax and Logic Meaning

Definition 1. *A* probabilistic abductive logic program (PALP) *is given by*

- *a set of predicate symbols, each with a fixed arity, distinguished into four disjoint classes,* abducibles*,* program defined*,* external *and* $\bot$*,*
- *for each abducible predicate* a/n*, a probability declaration of form* `abducible(`a`(_`$_1$`,...,_`$_n$`), `p`)`*, with* $0 < p < 1$*.*
- *A set of clauses of the form,* A_0 `:-`A_1`,...,`A_n*, of which the following kinds are possible,*
  - ordinary clauses *where* A_0 *is an atom of a program defined predicate and none of* $A_1, \ldots, A_n$*,* $n \geq 0$*, are* $\bot$*,*
  - integrity constraints *in which* $A_0 = \bot$ *and* $A_1, \ldots, A_n$*,* $n \geq 1$*, are abducible atoms.*

As usual, an arbitrary and infinite collection of function symbols, including constants, are assumed, and atoms are defined in the standard way. □

The relationship $\models$ refers to the usual, completion-based semantics for logic programs; for external predicates, we assume a semantics independently of the actual program, and without specifying further, an a priori defined truth value for $\models e$ is given for any ground external atom e. In practice, external predicates can be Prolog built-ins or defined by additional Prolog clauses, or constraints given either by a Prolog library or by additional CHR rules. We need to require that any call to an external predicate always succeeds at most once; for simplicity, we leave out externals defined as constraints from our formal considerations, but indicate in the text how they should be treated. The difference is basically that satisfiability of a constraint depends on the current execution state, which makes statements about correctness more complicated but adds no conceptual difficulties.

When Π is a PALP and we write $\Pi \models \cdots$ or $\Pi \cup \cdots \models \cdots$, we use Π to refer to completion of all clauses of P including the integrity constraints and (for brevity) a theory defining a meaning for all external predicates. When no ambiguity arises, a clause is usually an ordinary clause, and integrity constraints will be referred to as such.

The notation $[\![F_1, \ldots, F_n]\!]$, F_i being formulas, is taken as a shorthand for $\exists(F_1 \wedge \cdots \wedge F_n) \wedge \neg\bot$. Notice the following trivial properties,

$$[\![A \wedge B]\!] \equiv [\![A]\!] \wedge [\![B]\!] \tag{1}$$

$$[\![A \vee B]\!] \equiv [\![A]\!] \vee [\![B]\!] \tag{2}$$

which means that any standard distributive law that does not involve negation can be used for formulas within $[\![-]\!]$.

Definition 2. *A* query *or* goal *is a conjunction of non-$\bot$ atoms; a set of ground abducible atoms is called a* state*; a set of (not necessarily ground) abducible atoms is called a* state term*. In the context of a PALP Π, we say that state or state term S is* inconsistent *whenever $\Pi \cup \forall S \models \bot$ and otherwise* consistent*. For two separated state terms S_1, S_2, we say that S_1* subsumes *S_2 and that S_1 is* more general than *S_2 , whenever*

$$\models \exists S_1 \leftarrow \exists S_2. \tag{3}$$

Whenever S_1 subsumes S_2 and vice-versa, we say that they are equivalent.

Given a PALP Π and a query Q, an explanation *for Q is a state term E such that*

$$\Pi \cup \exists E \models [\![Q]\!] \tag{4}$$

An explanation E for Q is minimal *if it is not a subsumed by any other explanation for Q. A set of minimal and pairwise separated explanations $\mathbf{E} = \{E_1, \dots E_n\}$ for Q is* complete *whenever*

$$\Pi \models [\![Q]\!] \leftrightarrow \exists E_1 \vee \dots \vee \exists E_n. \tag{5}$$

□

In non-probabilistic abduction, a preference is often given to explanations with as few literals as possible, but this is not relevant here as we introduce a more precise measurement for explanations, namely their probabilities.

Lemma 1. *Whenever E is an explanation for Q in a program Π and X a set of abducible atoms, $E \cup X$ is an explanation for Q iff $E \cup X$ is consistent. Any explanation E for Q has a subset which is a minimal explanation for Q.* □

Lemma 2. *The complete set of minimal explanations $\{E_1, \dots, E_n\}$ for Q in a PALP Π is unique qua equivalence on individual explanations. When, furthermore, E is an arbitrary explanation for Q, it holds for some i, $1 \leq i \leq n$, that E_i subsumes E.* □

Lemma 3. *Let Q be a query to a PALP Π and $E_1, \dots, E_n$ consistent and pairwise separated state terms where E_i does not subsume E_j for any $i \neq j$. Whenever*

$$\Pi \models [\![Q]\!] \leftrightarrow \exists E_1 \vee \dots \vee \exists E_n. \tag{6}$$

it holds that $E_1, \dots, E_n$ comprise a complete set of explanations for Q. □

Example 1. Consider the following PALP, which we call Π_0.

```
abducible(a, 0.5).
abducible(b, 0.5).
abducible(c, 0.5).
p:- a, q.
q:- b.
q:- c.
⊥:- a,b.
```
(7)

We notice that $\{\mathtt{a}, \mathtt{c}\}$ is a minimal explanation for $\mathtt{p}$, and $\{\{\mathtt{a}, \mathtt{c}\}\}$ is complete. Other the other hand, we have that $\Pi_0 \cup \{\mathtt{a}, \mathtt{b}\} \models \mathtt{p}$ but since $\Pi_0 \cup \{\mathtt{a}, \mathtt{b}\} \models \bot$, we have $\Pi_0 \cup \{\mathtt{a}, \mathtt{b}\} \not\models [\![\mathtt{p}]\!]$. □

2.2 Probability Distributions for PALPs

A probabilistic model for a PALP Π is given by considering any ground abducible literal[1] A as a random variable with two outcomes, *true* with probability p and *false* with probability $1-p$, where p is the probability declared in Π for A. Any two such random variables are considered independent. We consider the outcome of the probabilistic experiment of giving values to all those variables as the state of those that come out as true. The joint distribution for a given PALP is defined formally as follows.

Definition 3. *For given PALP Π, the* probability distribution P_Π *is defined as follows.*

- $P_\Pi(true) = 1$
- *Whenever* $\mathtt{abducible(}A\mathtt{,}p\mathtt{)} \in \Pi$, *let* $P_\Pi(a) = p$ *for any ground instance* a *of* A.
- *Whenever* $\Pi \models A \leftrightarrow B$, *let* $P_\Pi(A) = P_\Pi(B)$.
- *Whenever* $P_\Pi(F) = p$, *let* $P_\Pi(\neg A) = 1 - p$.
- *Whenever* a *and* b *are two ground abducibles, let* $P_\Pi(a \wedge b) = P_\Pi(a) \times P_\Pi(b)$ *and* $P_\Pi(a \vee b) = P_\Pi(a) + P_\Pi(b) - P_\Pi(a \wedge B)$.
- *Whenever* A *has an infinite set of ground explanations* $E_1, E_2, \ldots$, *let* $P_\Pi(A) = \lim_{n\to\infty} P_\Pi(E1 \vee \cdots \vee E_n)$.

□

Example 2. Consider again the program of example 1. We notice that $P(\bot) = P(a, b) = P(a) \times P(b) = 0.25$ and thus $P(\neg\bot) = 0.75$. This means the only 75% of all states are relevant for the search for explanations for, say, $\mathtt{p}$.

The probability $P(\mathtt{p})$ is an uninteresting number as it counts also contributions from inconsistent states. The probability $P([\![\mathtt{p}]\!]) = 0.125$ measures $\mathtt{p}$ among all states, and gives here a lower figure than $P([\![\mathtt{p}]\!]|\neg\bot) = 0.167$ which measures among consistent states only. □

Finally, we notice a property, which can be utilized to produce best-first interpreters.

Proposition 1. *Whenever* S_1 *subsumes* S_2, *for two separated state terms* S_1, S_2, *it holds that*

$$P(S_1) \geq P(S_2). \tag{8}$$

□

[1] Notice that may indicates an infinity of random variables, when an abducible declarations contain variables. However, for any query to a well-behaved program, only a finite number of these are actually accessed, and the infinitely many remaining ones can be ignored.

3 Specifications of Auxiliary Predicates

The different query interpreters use a common collection of auxiliary predicates specified as follows; alternative implementations have been developed, an efficient one which assumes ground abducibles and a more general one which handles correctly also abducibles with variables.

We assume a context which includes a PALP so that we can refer to the notion of consistency and a probability distribution P.

`subsumes(`E_1`,`E_2`)` $\equiv E_1$ subsumes E_2, i.e., $\models \exists E_2 \rightarrow \exists E_1$, when E_1, E_2 are consistent and separate state terms.

`entailed(`A`,`E`)` $\equiv \models \forall(E \rightarrow A)$ when A is an abducible atom and E a consistent state term.

`extend(`A`,`E`,`$P(E)$`,`E'`,`$P(E')$`)` $\equiv \models \forall(E' \leftrightarrow A \wedge E)$ when A is an abducible atom and E, E' consistent state terms so that `entailed(`A`,`E`)` does not hold.

Notice the different usages of quantifiers. For `entailed/2` and `extend/5`, the presence of common variables in the arguments are significant, and variables may be bound later in the computation, whereas `subsumes/2` concerns explanations in different branches of computation (which, in fact, will have no variables in common).

The following predicate is used whenever an explanation may be affected by unifications, which may be a consequence of applying a rule of the given PALP or a executing call to an external predicate.

`recalculate(`E`,`E_1`,`$P(E_1)$`)` $\equiv \forall(E \leftrightarrow E_1)$, E_1 is in reduced form, when E and E_1 are consistent state terms.

We have introduced this predicate since it can be implemented quite efficiently by multiplying probabilities for the abducibles in E'; it does not seem feasible to analyze the detailed effect of a unification in order to reuse the previous probability.

Finally, we need the following renaming predicates in order to create alternative variants of a query when the execution splits in different branches for alternative clauses of the given PALP.

`rename(`T_1`,`T_2`)` $\equiv T_2$ is a variant of T_1 with new variables that are not used anywhere else.

4 Query Interpreters for Propositional Programs

We consider firstly a propositional version of probabilistic abductive logic programs (PPALPs), i.e., all predicates have arity 0. For simplicity we assume also that PPALPs contain no recursion, and that any non-abducible predicate appears as the head of at least one clause; furthermore, we exclude integrity constraints and external predicates, which means that there are no loops and failures to worry about.

Example 3. The following is a PPALP which introduces abducibles `a`, `b`, `c`, `d`, each with probability 0.5, and three clauses.

```
abducible(a, 0.5).
abducible(b, 0.5).
abducible(c, 0.5).
abducible(d, 0.5).                                    (9)
g:- a,b.
g:- c.
g:- c,d.
```

A set of minimal explanations for `g` with probabilities is given by $P(\mathtt{a},\mathtt{b}) = 0.25$, $P(\mathtt{c}) = 0.5$. Furthermore, $P(\mathtt{g}) = P(\mathtt{a},\mathtt{b}) + P(\mathtt{c}) - P(\mathtt{a},\mathtt{b},\mathtt{c}) = 0.5^2 + 0.5 - 0.5^3 = 0.625$. □

4.1 Translating PPALPs into All-Explanations Query Interpreters in CHR

Here we explain how any given PPALP Π can be translated into a CHR program Γ_Π, which serves as a query interpreter. Such an interpreter takes a query Q to Π as input and returns a final constraint store which contains a complete set of minimal explanation for Q in Π with their probabilities. The best-first interpreters and interpreters for more general classes of programs described later are all adaptation of what we show for PPALPs here.

We demonstrate the principles for compiling PPALPs into CHR for the program of example 3.

To find explanations for a goal such as `g`, we call the top-level predicate `explain([g])` which is defined as follows.

```
explain(G):- explain(G,[],1).                         (10)
```

The predicate `explain(`Q`,`E`,`p`)` is a CHR constraint governed by the rules given below; its meaning is that the query Q is what remains to be proven in order to find an explanation for the initial query; E is the partial explanation used so far in order to get from the initial query to Q, and p is the probability of E; Q is represented as a list of atomic goals. We do not need to consider the actual representation of explanations as the auxiliary predicates specified in section 3 provide an abstract datatype for them; the only assumption is that the empty explanation is represented as `[]`.

The following CHR rule interprets a query whose first subgoal is an abducible, adds it to the accumulating explanation if necessary (and adjusting propability accordingly) and emits a recursive call for the remaining part of the query.

```
explain( [A|G], E, P) <=> abducible(A,PA) |
   (entailed(A,E) -> explain(G, E, P)
     ;                                                (11)
     extend(A,E,P,E1,P1), explain(G, E1, P1) ).
```

Each collection of clauses defining a given predicate in the PPALP is translated into one CHR rule which produces new calls to `explain/3` for each clause. For our example program there is one such CHR rule.

```
explain( [g|G], E, P) <=>
   explain([a,b|G],E,P),
   explain([c|G],E,P),
   explain([c,d|G],E,P).
```
(12)

These clauses are sufficient to produce a complete set of explanations, represented as a final constraint store consisting of constraints `explain([],`E`,`$P(E)$`)` where E is an explanation for the initial query.

In order to remove non-minimal explanations, the following CHR rule is added as a first one to the interpreter program.[2]

```
explain([],E1,_) \ explain(_,E2,_) <=>
    subsumes(E1,E2) | true.
```
(13)

Notice that it may discard a branch early as soon as it can be seen that the possible explanations generated along that branch are deemed non-minimal.

To interpret the query `g` in the original PPALP, we can now pose the query `explain([g])` to the CHR program described above, which, in accordance with our expectations, yields the following final constraint store.

```
explain([],[c],0.5),
explain([],[a,b],0.25)
```
(14)

Notice that the constraint `explain([d],[c],0.5)` has appeared in the constraint store during the execution, but is has been discarded by the rule (13) and thus never executed until the end.

Lemma 4. *Let Π be a PPALP, Q a query, and Γ the translation of Π into a CHR program as described above in this section. Any constraint store which arises in the execution of* `explain(`Q`,[],1)` *in Γ is of the form*

$$\texttt{explain(}Q_1\texttt{,}E_1\texttt{,}p_1\texttt{)},\ldots,\ \texttt{explain(}Q_n\texttt{,}E_n\texttt{,}p_n\texttt{)} \tag{15}$$

where

$$\Pi \models Q \leftrightarrow ((Q_1 \wedge E_1) \vee \cdots \vee (Q_n \wedge E_n)) \tag{16}$$

and for all i, $1 \leq i \leq n$, $\Pi \models (Q_i \wedge E_i) \rightarrow Q$ and $P(E_i) = p_i$. □

Theorem 1. *Assume the setting of lemma 4. Whenever* `explain(`Q`)` *is posed as a query to Γ, the final constraint store is of the form*

$$\texttt{explain([],}E_1\texttt{,}p_1\texttt{)},\ldots,\ \texttt{explain([],}E_n\texttt{,}p_n\texttt{)} \tag{17}$$

where $E_1, \ldots, E_n$ comprise a complete set of minimal explanations for Q in Π, and all i, $1 \leq i \leq n$, $P(E_i) = p_i$. □

[2] Logically, rule (13) can be placed anywhere in the CHR program, but having it as the first rule makes it more effective in discarding irrelevant branches as early as possible.

4.2 Best-first query interpreters for PPALPs

For complex abductive problems it can be too cumbersome to calculate all possible minimal explanations, and instead we may want to calculate a minimal explanation with highest probability.

We can change the query interpreters shown so far, so they consider the constraint store as a priority queue of calls to `explain/3`, ordered by their current probabilities. During the process, we select the one with highest probability, allows it to make one step, and put back the derived calls; this continues until an explanation is found.

To implement this, we may replace `explain/3` by two other constraints `queue_explain/3` and `step_explain/3`. Whenever `queue_explain/3` is called, it means to enter a call into the queue; selecting a `queue_explain(`q`,`e`,`$P(e)`)` for execution is done by promoting it to another constraint `step_explain(`q`,`e`,`$P(e)`)`, which then makes one step for the first subgoal of q similarly to what we have seen above.

There will be at most one `step_explain/3` constraint in the store at a time, and it is selected either by an explicit call (when it is known by context that a particular constraint can be selected) or by an explicit search process. Searching the currently most probable partial explanation is done by posting a constraint `select_best/0` implemented by the following rules; `max_prob/1` is an auxiliary constraint used in the guard to check that the `queue_explain/3` constraint in focus actually is the best one.

```
queue_explain(G,E,P)#W, select_best <=> max_prob(P) |
    step_explain(G,E,P)
    pragma passive(W).
max_prob(P0), queue_explain(_,_,P1)#W <=> P0 < P1 | fail
    pragma passive(W).
max_prob(_) <=> true.
```
(18)

This is clearly not the most efficient way to implement a priority queue, but has been chosen here for the brevity of the code. See [22] for a more detailed study of priority queues in CHR.

We can extend this interpreter so it can generate more explanations in order of decreasing probabilities when requested by the user. This requires that we store solutions already printed out so that (partial) explanations subsumed by any of those can be discarded; to this end, we introduce an additional constraint `printed_explain/3` in order to avoid interference with the search for the currently best among non-printed, partial explanation.

We show the entire query interpreter which is a straightforward adaptation of the one shown in section 4.1; it encodes the same sample PPALP program as above (example 3).

```
explain(G):- step_explain([G],[],1).

printed_explain([],E1,_) \ queue_explain(_,E2,_) <=>
    subsumes(E1,E2) | true.

queue_explain(G,E,P)#W, select_best <=> max_prob(P) |
    step_explain(G,E,P)
    pragma passive(W).

step_explain([],E,P) <=> max_prob(P) |
    printed_explain([],E,P),
    write('Most probably solution: '), write(E),
    write(', P='), write(P),nl,
    ( user_wants_more -> select_best ; true ).
```

(19)

```
step_explain( [g|G], E, P) <=>
    queue_explain([a,b|G],E,P),
    queue_explain([c,d|G],E,P),
    step_explain([c|G],E,P). % select an arbitrary one

step_explain( [A|G], E, P) <=> abducible(A,PA) |
   (entailed(A,E) -> explain(G, E, P)
     ;
     extend(A,E,P,E1,P1), explain(G, E1, P1) ).

user_wants_more:-
    Ask user; if answer is y, succeed, otherwise fail.
```

(20)

The following shows part of the dialogue for the execution of the query `q` to the sample program.

```
| ?- explain(g).
Most probably solution: [c], P=0.5
Another and less probable explanation? y
Most probably solution: [a,b], P=0.25
```

(21)

Correctness of the best-first query interpreter can be stated and proved similarly to theorem 1 above. In fact a CHR derivation made by the best-first query interpreter corresponds to one possible derivation performed by the all-explanations query interpreter (given a nondeterministic operational semantics for CHR).

5 Programs with Variables, Unification, Integrity Constraints, and External Predicates

We now generalize the construction above to handle general PALPs, including parameterized abducibles, integrity constraints, and possibly external predicates.

The query interpreters for PPALPs of section 4.1 are straightforward to extend to handle variables. Whenever a non-abducible subgoal g with continuation c is rewritten into alternatives corresponding to clauses of the ALP, we produce a variant with new variables g', c' for each alternative; if g' unifies with the head of a clause, this alternative is continued, otherwise this branch is discarded (and thus avoiding failure in the overall process).

As already mentioned, the auxiliary predicates specified in section 3 are provided in two versions, an efficient one which aborts in case of nonground abducibles, and a more general and less efficient one which handles nonground abducibles in a correct way; both are given in appendix A.

Before going into the details, we explain how specific bindings to variables in the top-level query can be made part of the generated explanations, if desired. We may extend the interpreters with an extra argument for this, but we can also access the values by introducing a special abducible predicate for the purpose.[3]

```
abducible(value_of(_,_):1).                                    (22)
```

Stating now a query to a correct query interpreter (such as those introduced below) in the following way,

```
query([value_of('X',X), q(X)]),                                (23)
```

any explanation will be of the form $\{\texttt{value\_of('X',}v\texttt{)}\} \cup E_v$, where v is the value (if any; otherwise it is returned as a variable) bound to variable `X` in the construction of explanation E_v.

5.1 Unification and Failure

We illustrate the general principle by an example. Assume the predicate `p/1` is defined by the following clauses.

```
p(X):- q(X,Y), r(Y).
p(X):- a(X).                                                   (24)
p(1).
```

These clauses are compiled into the CHR rule (25) The last line shows handling of failure which in this case may arise when the variable `Xr3` has a ground value different from `1`. When all arguments in the head of a clause are distinct variables, these variables can be propagated into the body, and explicit unification and test for failure can be avoided. Notice for the last alternative, that renaming is suppressed since no further usages are made of the variables in the original query. The pattern `(`*test* `->` *continue* `; true)` means that a possible failure of *test* is absorbed, and the branch *continue* vanishes rather that provoking a failure

[3] Properly speaking, our definition of PALPs, def. 1, excludes abducibles with probability 1. This is for reason clarity only, as normally an abducible which is always true is not very interesting. However, there are no technical problems in handling such abducibles, so the `value_of/2` abducible will be handled correctly.

in the execution of the CHR rules (that would make the entire process fail); this technique is also used in [14, 6]. Recalculation of the probability in the last alternative is needed as the unification might have unified some variable in the explanation with a value, perhaps even making preciously different abducible atoms identical, thus possibly lowering the probability.

```
explain( [p(X)|G], E, P) <=>
    rename([p(X)|G]+E,[p(Xr1)|Gr1]+E1),
    explain([q(Xr1,Y),r(Y)|Gr1], E1, P),
    rename([p(X)|G]+E,[p(Xr2)|Gr2]+E2),
    explain([a(Xr2)|Gr2], E2, P),
    (X=1 ->
      recalculate(E,Er,Pr), explain(G, Er, Pr)
     ; true).
```
(25)

With the version of the auxiliaries that assumes always ground explanations (and aborts otherwise), the explanations need not be passed through the renaming and the call to `recalculate/3` can be left out.

The rule for accessing abducibles (11) is unchanged.

5.2 External predicates

External predicates are exported to the underlying Prolog+CHR system by the following rule; when placed following rules (25,11) there is no need to include a test that the predicate of the first subgoal (`X` below) actually is external. Possible failure of the external predicates is handled as described above, section 5.1.

```
explain([X|G], E, P) <=> true |
    (call(X) ->
      recalculate(E,Er,Pr), explain(G, Er, Pr)
     ; true).
```
(26)

All other parts of the query interpreters are unchanged, i.e., rules (10,11,13).

When external predicates are constraints of an external constraint solver, the renaming operation must also produce new copies of possible constraints pending on variables in the query argument being renamed. How these constraints may be accessed depends on that particular constraint solver. The clp(Q) and clp(R) constraint solver of SICStus Prolog [3, 24] include a predicate `projecting_assert`, which adds, in the body of a clause, the constraints pending on variables its argument. Appendix A.2 shows how this is incorporated into the renaming.

6 Related work

Abduction in logic programming without probabilities has attracted a lot of attention, and several algorithms, including as metainterpreters written in Prolog has been made; see [19, 11] for overview and references. We may emphasize an

early work by Console *et al* [9] from 1991, that explained abductive reasoning in terms of deductive reasoning in the completion of the abductive logic program. This principle was extended into an abstract procedure for abduction by Fung and Kowalski in 1997 [15], which inspired several implemented systems. Ignoring the probabilistic part of our own interpreters, they show similarity with the principle of [9] in the sense that we map abductive programs into a purely deductive paradigm, and each derivation step respects the program completion. Recent approaches to abductive logic programming [10, 12, 20] have added the interaction with specific, externally defined constraint solvers (but not with probabilities).

Abduction without probabilities has been approached using CHR, initially by [1] and later extended and applied in different context by [7, 5, 4, 8]; see [16] for an overview paper.

Different paradigms for prioritized, weighted and probabilistic abduction have been studied in different contexts, e.g., [2, 21, 23]. The mentioned approaches do not consider general integrity constraints and integration with constraint solvers. In [18], we generalize the approach presented in the present paper to a more general class of prioritized abductive programs. We shall refrain from giving detailed references to the tradition of Bayesian networks, which can be seen as a highly restricted case of probabilistic abduction (from a semantic point of view), but for which efficient methods exist that can handle very large data sets.

A Implementations of Auxiliary Predicates

We describe here the two alternative implementations for the auxiliary predicates specified in section 3, an effficient one for ground abducibles, and another one at more general one that can handle nonground abducibles.

A.1 For Ground Abducibles

Here we represent explanations as lists of ground abducibles sorted by Prolog's built on term ordering denoted `@<`.

```
subsumes(S1,S2):- fastsubset(S2).

fastsubset([],_).
fastsubset([X|Xs],[Y|Ys]):-
  X==Y -> fastsubset(Xs,Ys)
  ;  X @> Y -> fastsubset([X|Xs],Ys).

entailed(A,S):- fastmember(A,S).

fastmember(X,[Y|Ys]):-
  X==Y -> true
  ; X @> Y -> fastmember(X,Ys).

extend(A,S,P,S1,P1):-
```

```
  extend1(A,S,S1),
  \+ inconsistent(S1),
  abducible(A,PA), P1 is P*PA.

extend1(X,[],[X]).
extend1(X,[Y|Ys],[X,Y|Ys]):- X@<Y, !.
extend1(X,[Y|Ys],[Y|Ys1]):- extend1(X,Ys,Ys1).

% recalculate/3 not used here

rename(X,Y):- assert(aux(X)), retract(aux(Y)).
```

Inconsistency is defined specifically for the PALP at hand. Assume, as an example, that it contains the following integrity constraints.

```
  ⊥:- a, b.
  ⊥:- c(X), b(X).
```
(27)

Then the predicate is defined as follows.

```
  inconsistent(E):- subset([a,b],E).
  inconsistent(E):- subset([c(X),b(X)],E).

  subset([],_).
  subset([X|Xs],S):- member(X,S), subset(Xs,S).
```
(28)

A.2 For Nonground Abducibles and Constraints

Sideeffects in terms of unifications can occur which will destroy the term ordering within a list of nonground abducibles, so we use non-sorted lists instead. We show here a version which does not take into account possible delayed called or external constraints pending on the variables of the explanations and abducibles that are operated on. This is a bit tricky to add but involves no conceptual difficulties.

```
subsumes(S1,S2):-
   rename(S1,S1copy),
   rename(S2,S2sko), numbervars(S2sko,0,_),
   subset(S1copy,S2sko).

entailed(A,S):- \+ hard_member(B,S).

extend(A,S,P,[A|S],P1):-
  \+ inconsistent([A|S]),
  (ground(A) -> P=P1 ; abducible(A,PA), P1 is P*PA).

recalculate(E,E1,P1):-
    remove_dups(E,E1),
    prob(E1,P1).
```

```
remove_dups([],[]).

remove_dups([A|As], L):-
    hard_member(A,As) -> remove_dups(As, L)
    ; remove_dups(As, L1), L=[A|L1].

hard_member(A,[B|Bs]):-
    A==B -> true ; hard_member(A,Bs).

prob([],1).
prob([A|As],P):-
   \+ ground(A) -> prob(As,P)
   ; abducible(A,PA), prob(As,PAs), P is PA*PAs.

% inconsistent/2, as above
% subset/2 as above
```

The renaming operation must also produce new copies of the constraints pending on its input argument. For SICStus Prolog's clp(Q) and clp(R) constraint solvers [3, 24], this can be made as follows; check the references for further explanation.

```
rename(X,Y):-
    assert(aux(X)),retract(aux(Y)),
    projecting_assert(aux(X)),
    aux(Y), retract((aux(_):- _)).
```

Acknowledgement: This work is supported by the CONTROL project, funded by Danish Natural Science Research Council.

References

1. Slim Abdennadher and Henning Christiansen. An experimental CLP platform for integrity constraints and abduction. In *Proceedings of FQAS2000, Flexible Query Answering Systems: Advances in Soft Computing series*, pages 141–152. Physica-Verlag (Springer), 2000.
2. Eugene Charniak and Solomon Eyal Shimony. Cost-based abduction and map explanation. *Artificial Intelligence*, 66(2):345–374, 1994.
3. Christian Holzbaur. OFAI clp(q,r) Manual, Edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.
4. H. Christiansen and V. Dahl. Meaning in Context. In Anind Dey, Boicho Kokinov, David Leake, and Roy Turner, editors, *Proceedings of Fifth International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT-05)*, volume 3554 of *Lecture Notes in Artificial Intelligence*, pages 97–111, 2005.
5. Henning Christiansen. CHR Grammars. *Int'l Journal on Theory and Practice of Logic Programming*, 5(4-5):467–501, 2005.
6. Henning Christiansen. On the implementation of global abduction. In Katsumi Inoue, Ken Satoh, and Francesca Toni, editors, *CLIMA VII*, volume 4371 of *Lecture Notes in Computer Science*, pages 226–245. Springer, 2006.

7. Henning Christiansen and Verónica Dahl. Logic grammars for diagnosis and repair. *International Journal on Artificial Intelligence Tools*, 12(3):227–248, 2003.
8. Henning Christiansen and Verónica Dahl. HYPROLOG: A new logic programming language with assumptions and abduction. In Maurizio Gabbrielli and Gopal Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2005.
9. Luca Console, Daniele Theseider Dupré, and Pietro Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.
10. Marc Denecker and Antonis C. Kakas. Special issue: abductive logic programming. *Journal of Logic Programmming*, 44(1-3):1–4, 2000.
11. Marc Denecker and Antonis C. Kakas. Abduction in logic programming. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2407 of *Lecture Notes in Computer Science*, pages 402–436. Springer, 2002.
12. Ulrich Endriss, Paolo Mancarella, Fariba Sadri, Giacomo Terreni, and Francesca Toni. The ciff proof procedure for abductive logic programming with constraints. In José Júlio Alferes and João Alexandre Leite, editors, *JELIA*, volume 3229 of *Lecture Notes in Computer Science*, pages 31–43. Springer, 2004.
13. Thom Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
14. Thom W. Frühwirth and Christian Holzbaur. Source-to-source transformation for a class of expressive rules. In Francesco Buccafurri, editor, *APPIA-GULP-PRODE*, pages 386–397, 2003.
15. Tzee Ho Fung and Robert A. Kowalski. The iff proof procedure for abductive logic programming. *Journal of Logic Programmming*, 33(2):151–165, 1997.
16. Henning Christiansen. Executable specifications for hypotheses-based reasoning with Prolog and Constraint Handling Rules. *Journal of Applied Logic*; to appear, 2008.
17. Henning Christiansen. Implementing Probabilistic Abductive Logic Programming with Constraint Handling Rules. To appear, 2008.
18. Henning Christiansen. Prioritized abduction with CHR. To appear, 2008.
19. A.C. Kakas, R.A. Kowalski, and F. Toni. The role of abduction in logic programming. *Handbook of Logic in Artificial Intelligence and Logic Programming,* vol. 5, Gabbay, D.M, Hogger, C.J., Robinson, J.A., (eds.), Oxford University Press, pages 235–324, 1998.
20. Antonis C. Kakas, Bert Van Nuffelen, and Marc Denecker. A-system: Problem solving through abduction. In Bernhard Nebel, editor, *IJCAI*, pages 591–596. Morgan Kaufmann, 2001.
21. David Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artif. Intell.*, 94(1-2):7–56, 1997.
22. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In Michael Fink, Hans Tompits, and Stefan Woltran, editors, *WLP*, volume 1843-06-02 of *INFSYS Research Report*, pages 182–191. Technische Universität Wien, Austria, 2006.
23. Mark E. Stickel. A prolog-like inference system for computing minimum-cost abductive explanations in natural-language interpretation. *Annals of Mathematics and Artificial Intelligence*, 4:89–105, 1991.
24. Swedish Institute of Computer Science. SICStus Prolog user's manual, Version 4.0.2. Most recent version available at `http://www.sics.se/isl`, 2007.