Implementing Probabilistic Abductive Logic Programming with 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 transformed 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 have been developed. Abductive reasoning means to search for missing world facts, which can explain observations of the state of affairs. Diagnosis in medicine and fault finding in mechanical or virtual systems are some of the obvious applications.

In general, abductive reasoning based on logic programs tends to provide too many and often strange explanations, and integrity constraints, which are formalized conditions which must hold in the possible worlds expressed by different explanations, can be applied for ruling out some of those. Another important issue is that explanations should be minimal, in the sense that they do not contain information which is not necessary in order to explain the observation.

Finally, adding probabilities to a knowledge representation formalism provides a way to prioritize among different explanations, giving a measurement of which one is better (i.e., more probable) than others. Probabilities may furthermore be applied to optimize the search for explanations, always going in

the most probable direction, so that investigation of less probable alternatives is suppressed or postponed.

While a lot of research has been made, and several systems developed, in logic programming based settings, far less work has been done in combining with probabilities. We suggest here an implementation of abduction in probabilistic logic programs in Constraint Handling Rules (CHR) which serves two purposes. Firstly, it overcomes certain limitations of earlier work and provides a very flexible architecture, which points forward to different extensions such as interaction with a non-monotonically evolving world. Secondly, it demonstrates CHR's suitability as a metalanguage for implementing advanced reasoning patterns, which is a direction we have pursued also in earlier work.

In fact, the major part of the CHR rules that we present expose in a clear and abstract way, the strategies used in the search for minimal, probabilistic explanations. In this way, CHR is experienced as a unique metaprogramming language for an overall methodology, which is to apply CHR's constraint store as a pool of pending computational processes, which collectively maintains the meaning of the observation posted as a query to the system, and where each process gradually moves towards an explanation, perhaps splitting into other processes along the way. These processes can be run either exhaustively in the arbitrary order provided by the underlying CHR system, or using an explicit scheduling policy such as best-first.

In addition to provide working implementations, our work may also be useful in a pedagogical context (teaching students *what* is and *how* to make probabilistic abduction), and finally it may provide executable specifications for detailed and very efficient implementations in low-level language such as C. In the present paper, we present implementations in terms of concrete and executable code, with only very few details left out. Notice that in some cases, we have given priority to brevity of the code rather than ultimate efficiency.

Overview

Section 2 introduces the language of Probabilistic Abductive Logic Programs (PALPs) with its logic and probabilistic semantics. PALP programs include declarations of abducibles with prior probabilities, integrity constraints, calls to external predicates, but no negation. External predicates can be a defined in Prolog or be constraints for which a solver is given, by additional CHR rules or otherwise.

In section 3, we provide specifications of auxiliary predicates used in our subsequent implementations, which define, so to speak, an abstract datatype for explanations. We consider two alternative implementations (details in appendix B), a straightforward and efficient one which aborts in case of nonground abducibles, and another one with full generality.

Our implementation of PALPs is given as a systematic transformation of a given program into a CHR program which, then, serves as a query interpreter. Section 4 explains this transformation for a propositional subset of PALPs in

order to outline the basic principles; two implementations are given, an all-solutions and a best-first version. Section 5 adds the remaining details to provide implementations for the full PALP language.

We do not specify in detail the semantics used for CHR in our proofs, but assume a semantics "as usual", given by [24]; in our proofs, we argue in a semi-formal style in terms of an operational semantics for CHR which in most cases considers it as a nondeterministic rewriting system, and occasionally we need to refer to CHR's sequential search for rules to apply and its left-to-right execution of rule bodies (cf. [22]).

Section 6 indicates further extensions and optimizations, firstly inspired by Dijkstra's shortest path algorithm [21] which is relevant in cases where the residual query in each branch is a single atom, and secondly, by using simplification techniques [33, 18] to speed up integrity checking. Finally, we consider the addition of a limited form of negation and we can argue that a logically more satisfactory version of negation is difficult to handle probabilistically.

Section 7 provides two fully developed example PALP programs, including diagnosis and finding most the probable path in a network. Both are implemented in CHR using best-first search, and the second one shows also the Dijkstra optimization indicated above.

The final section 8 provides for a summary, an overview of related work, and perspectives for applications and extensions of the present work.

2 Probabilistic Abductive Logic Programs

2.1 Syntax and Logic Meaning

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

- *a set of predicate symbols, each with a fixed arity, distinguished into four disjoint classes,* abducibles*,* program defined*,* external *and* $\perp$*,*
- *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 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* $\perp$*,*
  - integrity constraints *in which* $A_0 = \perp$ *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 defined in the standard way. □

Notice that $\perp$ is a distinguished predicate rather that a representation of falsity. The relationship $\models$ refers to the usual, completion-based semantics for logic programs [37, 31]; for external predicates, we assume a semantics independently of

the actual program, and without specifying further, an 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 no ambiguity arises, a clause is usually an ordinary clause, and integrity constraints will be referred to as such. In the context of a PALP Π, a formula is called *basic*, if it can be rewritten into an equivalent form using the equivalences defined by the clauses of Π, consisting of conjunctions, disjunctions, negations and a finite number of ground abducible atoms and $\perp$. In the following we refer to different terms or formulas being *separated* meaning that they have no variables in common.

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

$$[\![A \wedge B]\!] \equiv [\![A]\!] \wedge [\![B]\!] \quad \text{for separated formulas } A \text{ and } B \tag{1}$$

$$[\![A \vee B]\!] \equiv [\![A]\!] \vee [\![B]\!] \quad \text{for arbitrary formulas } A \text{ and } B. \tag{2}$$

Example 1. Consider the following PALP.

```
abducible(some(_),0.1).
some_nat:- some(N), nat(N).
nat(0).                                              (3)
nat(s(N)):- nat(N).
loop(N):- some(N), loop(s(N)).
```

Here formulas `some_nat` and `loop(0)` are not basic; `nat(s(s(0)))` is basic. It is well-known that natural numbers can be represented by zero and a successor function, and that addition and multiplication can be implemented by a logic program. For subsequent examples, we assume the program above extended with for arithmetic and a predicate `sequence/1` with the following properties; the actual and lengthy definition is left out and n is used as a convenient writing of $\mathtt{s}^n\mathtt{(0)}$.

$$\begin{array}{l}
\mathtt{sequence(1)} \leftrightarrow \mathtt{some(1)} \\
\mathtt{sequence(2)} \leftrightarrow \mathtt{some(2),\ some(3)} \\
\mathtt{sequence(3)} \leftrightarrow \mathtt{some(4),\ some(5),\ some(6)} \\
\vdots \\
\mathtt{sequence(}n\mathtt{)} \leftrightarrow \mathtt{some(}\frac{(n-1)\times n}{2}+1\mathtt{),\ \ldots,\ some(}\frac{n\times(n+1)}{2}\mathtt{)} \\
\vdots
\end{array} \tag{4}$$

□

Definition 2. *A* query *or* goal *is a conjunction of non-$\bot$ atoms; a finite set (or conjunction) of ground abducible atoms is called a* state*; a finite 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*

$$\models \exists S_1 \leftarrow \exists S_2. \tag{5}$$

Whenever S_1 subsumes S_2 and vice-versa, we say that they are equivalent*; if S_1 subsumes S_2 and they are not equivalent, we say that S_1* strictly subsumes *S_2; if neither S_1 subsumes S_2 nor the reverse, we say that they are* incompatible*.*

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

$$\Pi \cup \exists E \models [\![Q]\!] \tag{6}$$

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

$$\Pi \models [\![Q]\!] \leftrightarrow \exists E_1 \vee \cdots \vee \exists E_n. \tag{7}$$

□

In practice, an answer for a query to an abductive logic program may include, in addition to the explanation as defined above, also a variable substitution and a set of normalized constraints of any external constraint solver applied. These details, which are straightforward to add, are left out for simplicity.

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

Example 2. Explanation {`a(X)`} subsumes {`a(1)`} as well as {`a(1)`, `a(2)`}.

Explanation {`a(X)`, `a(1)`} is equivalent to {`a(1)`}. However, explanations are built in an incremental way during the execution of a program (as explained later in this paper), in which variables may be quantified and bound at different levels. In the example, {`a(X)`, `a(1)`} as a partial explanation may be affected by `X=2` and lead to final explanation {`a(2)`, `a(1)`}. In order words, the replacement of one explanation by a smaller, equivalent one is only relevant for a final explanation to a query. □

Example 3. Consider again the PALP of example 1 above. The query `some_nat(N)` has minimal explanations {`some(0)`}, {`some(1)`}, ...; `loop(N)` has no explanations; `sequence(N)` has explanations {`some(1)`}, {`some(2)`, `some(3)`}, {`some(4)`, `some(5)`, `some(6)`}, ... □

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.* □

Proof. Trivial. □

Lemma 2. *The complete set of minimal explanations* $\{E_1, \ldots, E_n\}$ *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. □

Proof. See appendix A. □

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

$$\Pi \models [\![Q]\!] \leftrightarrow \exists E_1 \vee \cdots \vee \exists E_n. \tag{8}$$

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

Proof. See appendix A. □

Example 4. Consider the following PALP, which we call Π_0.

```
abducible(a, 0.5).
abducible(b, 0.5).
abducible(c, 0.5).
p:- a, q.                                  (9)
q:- b.
q:- c.
⊥:- a,b.
```

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 it is not an explanation as $\Pi_0 \cup \{\mathtt{a}, \mathtt{b}\} \models \bot$ and thus $\Pi_0 \cup \{\mathtt{a}, \mathtt{b}\} \not\models [\![\mathtt{p}]\!]$. □

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.*

[1] Notice that this may indicate 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.

- $P_\Pi(true) = 1$
- *Whenever* `abducible(`A`,`p`)` $\in \Pi$*, 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* $P_\Pi(A) = p$*, let* $P_\Pi(\neg A) = 1 - p$*.*
- *Whenever* a *and* b *are two distinct 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* $P_\Pi(A) = \lim_{n\to\infty} P_\Pi(E1 \vee \cdots \vee E_n)$*.* □

We observe, for the last case of the definition, that we need only consider minimal explanations, and that this part may overlap, but is not in conflict, with the other cases. Notice the following properties of the probability distribution. Whenever the program Π is clear from context, we may write P instead of P_Π.

Proposition 1. *Let* Π *be a PALP and* P_Π *its probability distribution.*

- *Whenever* A *is a nonground abducible atom,* $P_\Pi(\forall A) = 0$ *and* $P_\Pi(\exists A) = 1$*.*
- *For any basic formula* F *over* Π*,* $P(F) = 1$ *iff* $\Pi \models F$*.*
- *For any formula* F *over* Π*,* $P(F) = 0$ *iff* $\Pi \not\models F$*.*
- *For any formula* F *over* Π*,* $P(F) > 0$ *iff* $\Pi \cup S \models F$ *for some state* S*.* □

The restriction to basic terms is essential in the second case. For example, when $a/1$ is an abducible predicate, we have that $P_\Pi(\exists x\, a(x)) = 1$ but not necessarily $\Pi \models \exists x\, a(x)$. To see this, assume $\emptyset$ is a model Π, but $\emptyset$ is not a model of $\exists x\, a(x)$. Notice that the exclusion of probabilities 0 and 1 for abducibles is essential for the proposition.

Example 5. Consider the PALP of example 1. Here we get the following examples of probabilities for non-basic formulas.

$$P(\exists n \texttt{some_nat}(n)) = 1$$
$$P(\exists n \texttt{loop}(n)) = 0$$
$$P(\exists n \texttt{sequence}(n)) = p$$
$$\text{where } 0.1 < p < 0.1 + (0.1)^2 + (0.1)^3 + \cdots = 0.1111\cdots$$

The last example indicates that the limit construction may give a sum different from one or zero. This conclusion is based on formula (17) below. □

As is customary in formulas of probability theory, comma is used interchangeably with $\wedge$. Whenever F is a formula with free variables, we let $P(F)$ be a shorthand for $P(\exists F)$.

It is crucial for defining a probability distribution with reasonable properties, that $\perp$ is defined as a special predicate rather that falsity; using falsity would mean that a set of integrity constraints implied a complicated set of dependencies among the random variables (i.e., they were no longer independent).

Example 6. . Consider again the program of example 4. 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, `p`.

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

In the example, we indicated that $P([\![Q]\!]|\neg\bot)$ for some query Q may be more appropriate than $P([\![Q]\!])$ to characterize Q, but we should be aware that $P([\![Q]\!])$ is sufficient for comparing the relative order of probabilities, as the two measures are proportional:

$$P([\![Q]\!]|\neg\bot) = \frac{P([\![Q]\!], \neg\bot)}{P(\neg\bot)} = \frac{P([\![Q]\!])}{P(\neg\bot)}. \tag{10}$$

Notice that the introduction of integrity constraints in probability distribution has an interesting effect on observed probabilities of abducibles.

Example 7. We consider the program of examples 4 and 6. While $P(a) = 0.5$ according to its declaration, we have the following since state $\{a, b\}$ is inconsistent.

$$P([\![a]\!]|\neg\bot) = \frac{P([\![a]\!])}{P(\neg\bot)} = \frac{P(a \wedge \neg b)}{P(a \wedge \neg b) + P(\neg a \wedge b) + P(\neg a \wedge \neg b)} \tag{11}$$

$$= \frac{0.25}{0.25 + 0.25 + 0.25} = 1/3. \tag{12}$$

In other words, the restriction to consistent states modifies the probability abducibles. An integrity constraint such as `⊥:- a,b` does not conflict with the basic assumption of a and b being independent. However, $[\![a]\!]$ and $[\![b]\!]$ becomes dependent. □

The following observations may help to simplify the notation.

$$P([\![F]\!]) = 0 \quad \text{whenever} \quad \Pi \cup F \models \bot \tag{13}$$

$$P(F) = P([\![F]\!]) \quad \text{whenever} \quad \Pi \cup F \not\models \bot \tag{14}$$

Especially when F is a set of abducibles $\{a_1, ..., a_n\}$, we can write $[\![a_1, ..., a_n]\!]$ in a probabilistic formula to give it same weight as F when consistent (as are, e.g., explanations) and 0 otherwise. We notice the following trivial properties.

$$P([\![A \wedge B]\!]) = P([\![A]\!] \wedge [\![B]\!]) \quad \text{for separated formulas } A \text{ and } B \tag{15}$$

$$P([\![A \vee B]\!]) = P([\![A]\!] \vee [\![B]\!]) \quad \text{for arbitrary formulas } A \text{ and } B. \tag{16}$$

Notice especially, when $E_1, \ldots, E_n$ are separated state terms that we have the following.

$$
\begin{aligned}
P(E_1 \vee \ldots \vee E_n) = & \; P(E_1) + \cdots + P(E_n) \\
& - \textstyle\sum_{1 \le i_1 < i_2 \le n} P(E_{i_1}, E_{i_2}) \\
& + \textstyle\sum_{1 \le i_1 < j_2 < j_3 \le n} P(E_{i_1}, E_{i_2}, E_{i_3}) \\
& \vdots \\
& + (-1)^{k+1} \textstyle\sum_{1 \le i_1 < \cdots < i_k \le n} P(E_{i_1}, \ldots, E_{i_k}) \\
& \vdots \\
& + (-1)^{n+1} P(E_1, \ldots E_n)
\end{aligned}
\tag{17}
$$

$$
\begin{aligned}
P([\![E_1 \vee \ldots \vee E_n]\!]) = & \; P([\![E_1]\!]) + \cdots + P([\![E_n]\!]) \\
& - \textstyle\sum_{1 \le i_1 < i_2 \le n} P([\![E_{i_1}, E_{i_2}]\!]) \\
& + \textstyle\sum_{1 \le i_1 < j_2 < j_3 \le n} P([\![E_{i_1}, E_{i_2}, E_{i_3}]\!]) \\
& \vdots \\
& + (-1)^{k+1} \textstyle\sum_{1 \le i_1 < \cdots < i_k \le n} P([\![E_{i_1}, \ldots, E_{i_k}]\!]) \\
& \vdots \\
& + (-1)^{n+1} P([\![E_1, \ldots E_n]\!])
\end{aligned}
\tag{18}
$$

When using (18), we need for each summand $P([\![E_{i_1}, \ldots, E_{i_k}]\!])$, to check if the state comprised by $E_{i_1}, \ldots, E_{i_k}$ together with the integrity constraints can prove $\perp$ in which case the result is 0; otherwise, duplicates are removed and probabilities for the abducibles are multiplied.

The following propositions and observation indicate relationships between probabilities and subsumption.

Proposition 2. *Let S_1 and S_2 be state terms. Whenever S_1 subsumes S_2 it holds that $P(S_1) \ge P(S_2)$.*

When, furthermore, S_1 and S_2 are ground and subsumption is strict, it holds that $P(S_1) > P(S_2)$. □

Proposition 3. *Let S_1 and S_2 be ground state terms with $P(S_1) \ge P(S_2)$; then either S_1 subsumes S_2 or they are incompatible.*

When $P(S_1) > P(S_2)$, either S_1 strictly subsumes S_2 or they are incompatible. □

The first part of proposition 3 does not hold for nonground state terms. For example, if $a(-)$ and b are abducible, we have with $S_1 = \{a(_), b\}$ and $S_2 = \{b\}$ that $P(S_1) = P(S_2) = P(b)$, but S_1 does not subsume S_2.

3 Specifications of Auxiliary Predicates

The different query interpreters use a common collection of auxiliary predicates specified as follows; alternative implementations are shown appendix B.

We do not need to specify a representation for explanations here but we assume there is a notion of a *reduced form* of representations; we anticipate representations as lists of abducible literals, and the reduced form meaning no such literals entailed by others. From a logical point of view, the reduced form is not interesting, but is useful for efficiency and when presenting final explanations to the user. 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.

`normalize_final(`E_1`,`E_2`)` $\equiv$ E_2 is a normalized explanation such that E_1 and E_2 are equivalent.

Notice the different usages of quantifiers. For `entailed/2` and `extend/5`, the presence of common variables in the arguments is significant, and variables may be bound later in the computation, whereas `subsumes/2` concerns different final explanations arising in different branches of computation; compare with example 2.

The `normalize_final/2` predicate is logically redundant but is used to provide an intuitively more pleasing appearance of final explanations. We can illustrate the purpose, referring to example 2, above. Here it was argued that {`a(X)`, `a(1)`} is equivalent to {`a(1)`} and also that it is incorrect to replace {`a(X)`, `a(1)`} by the smaller one during the execution as `X` might be bound to some value. However, in the case {`a(X)`, `a(1)`} is recognized as an explanation for the initial query, the situation is different; there is no partial query left to manipulate `X`, so we can now replace it by the smaller and logically equivalent {`a(1)`}. We leave the predicate out in the detailed descriptions of the interpreters below as this is anyhow trivial to add and has no influence on the correctness statements.

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 executing a 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_1. It very seldom pays off

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.

Be aware that these predicates, as specified only works when external predicates exclude constraints of delayed calls. To include constraints, subsumption needs to be defined as $\Sigma \cup \Delta \models \exists E_2 \rightarrow \exists E_1$ where is Σ refers to the current execution state and Δ gives the semantics of the underlying constraint solver. The other predicates above that refer to $\models$ should be extended in similar ways, and `rename` must also add constraints to the state whenever variables in the input argument are covered by constraints. More details and examples are discussed in section 5.3 below.

As shown in the appendix, the implementation of subsumption and entailment can be greatly simplified if it can be guaranteed that the explanations always are ground. In that case, explanations can be represented as lists sorted by Prolog's term ordering (denoted `@<`) and subsumption test becomes an efficient sublist test for sorted lists (see appendix B). It is possible to define syntactic restrictions to ensure that abducibles always are ground so that the efficient implementation can be used, but our ground version uses runtime checks instead.

Lists of nonground abducibles have, furthermore, also the complication that a unification induced by a rule application or external predicate can destroy the sortedness of a list as well as making elements equal (more generally, making some elements subsumed by others).

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 8. 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).                                        (19)
g:- a,b.
g:- c.
g:- c,d.
```

A set of minimal explanations for g with probabilities is given by $P(\mathtt{a},\mathtt{b}) = 0.25$, $P(\mathtt{c}) = 0.5$. Using (17), we get $P(\mathtt{g}) = 0.5^2 + 0.5 - 0.5^3 = 0.625$. □

4.1 Transforming PPALPs into All-Explanations Query Interpreters in CHR

Here we explain how any given PPALP Π can be transformed 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 8.

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).                                        (20)
```

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 adjusts the probability 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)
     ;                                                               (21)
     extend(A,E,P,E1,P1), explain(G, E1, P1) ).
```

Each collection of clauses defining a given predicate in the PPALP is transformed 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),                                               (22)
   explain([c,d|G],E,P).
```

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.
```
(23)

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)
```
(24)

Notice that the constraint `explain([d],[c],0.5)` has appeared in the constraint store during the execution, but discarded by rule (23) and thus never executed until the end.

Lemma 4. *Let Π be a PPALP, Q a query, and Γ the transformation 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*

$$\texttt{explain(}Q_1\texttt{,}E_1\texttt{,}p_1\texttt{)},\ldots,\ \texttt{explain(}Q_n\texttt{,}E_n\texttt{,}p_n\texttt{)} \tag{25}$$

where

$$\Pi \models Q \leftrightarrow ((Q_1 \wedge E_1) \vee \cdots \vee (Q_n \wedge E_n)) \tag{26}$$

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

Proof. See appendix A. □

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*

$$\texttt{explain([],}E_1\texttt{,}p_1\texttt{)},\ldots,\ \texttt{explain([],}E_n\texttt{,}p_n\texttt{)} \tag{27}$$

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

Proof. See appendix A. □

[2] Logically, rule (23) 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 Conditional Probabilities

For a typical abductive problem, the probability of a given explanation may be very small and not very informative to the user. It may be more interesting to have the interpreter produce instead the conditional probability of each explanation E given the observation Q (i.e., the initial query), which is given as follows.

$$P(E|\llbracket Q\rrbracket) = \frac{P(E,\llbracket Q\rrbracket)}{P(\llbracket Q\rrbracket)} = \frac{P(E)}{P(\llbracket Q\rrbracket)} \tag{28}$$

Probabilities $P(E)$ are those calculated by the CHR program shown above, and $P(\llbracket Q\rrbracket)$ can be calculated from the final constraint store based on formula (17) (or (18) when we generalize to PALPs). In the example, we get $P(\mathtt{g}) = 0.625$ and thus, with the hinted extensions to the program, the following final constraint store.

```
explain_conditional([],[c],0.8),
explain_conditional([],[a,b],0.4)
```
(29)

Notice that the sum of these probabilities is > 1, which comes from the fact that both minimal explanations subsumes the non-minimal `[a,b,c]`, which has conditional probability $0.5^3/0.625 = 0.2$.

Whenever an abdicible a appears in more that one explanation, it may be interesting to calculate the probability of a given the observation.

$$P(\llbracket a\rrbracket|\llbracket Q\rrbracket) = \frac{P(\llbracket a\rrbracket,\llbracket Q\rrbracket)}{P(\llbracket Q\rrbracket)} = \frac{P(\llbracket a,Q\rrbracket)}{P(\llbracket Q\rrbracket)} \tag{30}$$

This can be found by first calculating $P(\llbracket Q\rrbracket)$ as above and then $P(\llbracket a,Q\rrbracket)$. However, with a bit of programming, it is be possible to obtain the value of $P(\llbracket a,Q\rrbracket)$ from the final constraint store used for finding $P(\llbracket Q\rrbracket)$, by summing up probabilities for the explanations that include a (or, in the general case, entail a).

4.3 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      (31)
    pragma passive(W).
max_prob(_) <=> true.
```

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 [42, 30] for more detailed studies of priority queues in CHR. Notice, that while constraints in the guard of a CHR rule may lead to dubious semantics, the call to `max_prob` in (31) makes sense as it does not change the constraint store or bind variables; it is handled sensibly by most CHR implementations.

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 8).

```
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)                                        (32)
    pragma passive(W).

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

```
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)                        (33)
     ;
     extend(A,E,P,E1,P1), explain(G, E1, P1) ).

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

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                     (34)
Most probably solution: [a,b], P=0.25
```

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).

For any solution found by the query interpreter, it is possible to provide an estimate[3] of the probability of the observation (in the example: g) and thus of the conditional probabilities considered in section 4.2 above. Assume that the query interpreter at a given stage of executing a query Q prints a minimal explanation E_k and that it has already printed $E_1, \ldots, E_{k-1}$, and let $E_{k+1}, \ldots, E_n$ be the remaining partial explanations in the store. Then we have

$$P(E_1, \ldots, E_k) \leq P(Q) \leq P(E_1, \ldots, E_n) \tag{35}$$

The probabilities defining the upper and lower limits can be calculated from the current constraint store based on formula (17).

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.

[3] This is inspired by [34]; our formula is a bit different from that of [34] since the basic assumptions are different.

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 PALP, 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 B.

5.1 Variables in Queries and Abducibles

Bindings made to variables an a query during its execution should be reported to the user. 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.

```
abducible(value_of(_,_),1).                                    (36)
```

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)]),                                (37)
```

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.

Notice that we defined abducibles earlier to have probabilities strictly less that 1 in order to have proposition 1. However, as `value_of` atoms are expected to be posted in the top-level query only and will remain fixed (but possibly affected by unifications) throughout the execution and with probability 1, it does not itself affect the probabilities of the explanations. The intuition that variable bindings add additional commitments is reflected in the subsumbtion hierarchy.

Example 9. Let $E_1 = \{\texttt{value_of('X',X)}, \texttt{a(X)}\}$ and $E_2 = \{\texttt{value_of('X',1)}, \texttt{a(1)}\}$ where `a` is an abducible predicate declared with probability 0.5. Then E_1 subsumes E_2 and $P(E_1) = 1 > 0.5 = P(E_2)$. □

5.2 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).                                                   (38)
p(1).
```

These clauses are compiled into the CHR rule (39) below; notice for a variable in the head of a clause, that we can propagate this variable into the body rather than performing an explicit unification; when all arguments in the head are variables, the unification is deemed to succeed, so a test for failure can be omitted. The last line shows handling of failure which in this case may arise when the variable `Xr3` has a ground value different from `1`. 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 in the execution of the CHR rules (that would make the entire process fail); this technique is also used in [25, 12]. Recalculation of the probability in the last alternative is needed as the unification might have unified some variable in the explanation with a value, 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),                          (39)
    (X=1 ->
      recalculate(E,Er,Pr), explain(G, Er, Pr)
     ; true).
```

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 (21) is unchanged.

An aside Remark on Splitting by Unification of Abducible: Aiming at explanations that are minimal in the number of abducible atoms, the majority of non-probabilistic abduction methods [28, 20] tries to unify a new abducible with existing ones if possible. However, in order not to sacrifice completeness, two brances of computation are initiated in each such case. For example if $a(s)$ is added to a partial explanation $\{a(t), \cdots\}$, one branch may continue after unifying s and t, with $\{a(t), \cdots\}$, and another one with $\{a(s), a(t), \cdots\}$ with the additional constraint that s and t must remain different. Our notion of minimality is based on subsumption and we avoid this splitting into two brances, and even we produce minimal explanations.

We have, in fact, two objections to the splitting approach; first of all conceptually since the unification of the two abducibles above indicates a commitment which is not grounded for in the knowledge base (see a detailed argument in [13]), and secondly, it may result in an exponential explosion in the number of brances that needs to be investigated.

5.3 External predicates

External predicates are exported to the underlying Prolog+CHR system by the following rule; when placed following rules (39,21), 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 5.2.

```
explain([X|G], E, P) <=> true |
    (call(X) ->
      recalculate(E,Er,Pr), explain(G,Er,Pr)
     ; true).                                          (40)
```

All other parts of the query interpreters are unchanged, i.e., rules (20,21,23).

In case of external predicates that use constraints or delays, we need to have the renaming of the current query produce new versions of constraints and other delayed calls pending on the variables in the query. Implementing a generalized renaming predicate that takes care of delayed call is quite straightforward provided that facilities are available for getting access to the delayed calls pending on specific variables.

Example 10. SICStus Prolog [43] includes a delaying predicate for non-equality, `dif/2`. Consider the case when there is a delayed call `dif(X,7)` for the variable `X` occurring in a query `[p(X),...]`. When this query is renamed into, say `[p(X1),...]`, we need also produce the new variant `dif(X1,7)` of the delayed call in order to provide a correct semantics.

The SICStus built-in predicate `frozen(X,C)` will assign to C a representation of all calls delayed on variable X, including `C=prolog:dif(X,7)` in the example above. The delayed calls can now have their variables renamed simultaneously with the query, and the resulting variant calls, say `dif(X1,7)`, can be entered into the program state simply by calling them. In this way the semantics is preserved in the copied query. □

Example 11. The `clpr` and `clpq` libraries of SICStus Prolog [43, 9] provide constraint solvers over real, resp., rational numbers, which can be used as external predicates in PALP. It provides a predicate `projecting_assert` by means of which a clause capturing the constraints on indicated variables can be created dynamically. Such a clause can be used in a straightforward way to produce the desired variants of constraints. We illustrate its use by an example; the curly brackets indicate the syntax for calling the constraint solver. Executing

```
{X=Y+Z}, projecting_assert(aux(p(X,Y,Z))).             (41)
```

creates a clause equivalent with the following,

```
aux(p(X,Y,Z)):- {X=Y+Z}.                               (42)
```

Calling this predicate with new arguments can set up the relevant constraints. The renaming predicate in appendix B is defined in the following standard way,

```
rename(X,Y):- assert(aux(X)),retract(aux(Y)).          (43)
```

and we can modify it for `clpr` and `clpq` as follows.

```
rename(X,Y):-
    assert(aux(X)),retract(aux(Y)),
    projecting_assert(aux(X)),                    (44)
    aux(Y), retract((aux(_):- _)).
```

No more adjustments are needed to incorporate these constraint solvers. □

We have not developed extended definitions (nor implementations) of subsumption and entailment that takes external constraints into account. For example, explanations including constraints {`a(X)`,{`X>7`}} and {`a(X)`} are considered equally good; intuitively, the last one should be preferred by a best first interpreter.

5.4 Correctness of the All-Explanations Query Interpreter for a PALP

To sum up, the PALPs interpreters are similar to those given for PPALPs in section 4 except that rules of form (39) replaces those of form (22), and that (40) is added.

Lemma 5. *Let Π be a PALP, Q a query, and Γ the transformation of Π into a CHR program, including auxiliary definitions, as described above in sections 5.2–5.3. Any constraint store which arises in the execution of* `explain(`Q`,[],1)` *in Γ is of the form*

$$\texttt{explain(}Q_1\texttt{,}E_1\texttt{,}p_1\texttt{)},\ldots,\ \texttt{explain(}Q_n\texttt{,}E_n\texttt{,}p_n\texttt{)} \tag{45}$$

where $Q_1+E_1,\ldots,Q_n+E_n$ are pairwise separate, and

$$\Pi \models [\![Q]\!] \leftrightarrow [\![Q_1,E_1]\!] \vee \cdots \vee [\![Q_n,E_n]\!] \tag{46}$$

and for all i, $1 \le i \le n$, $\Pi \models [\![Q_i,E_i]\!] \rightarrow [\![Q]\!]$ and $P(E_i)=p_i$. □

Proof. See appendix A. □

Theorem 2. *Assume the setting of lemma 5. Whenever* `explain(`Q`)` *is posed as a query to Γ, and the derivation terminates without error messages, the final constraint store is of the form*

$$\texttt{explain([],}E_1\texttt{,}p_1\texttt{)},\ldots,\ \texttt{explain([],}E_n\texttt{,}p_n\texttt{)} \tag{47}$$

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

Proof. See appendix A. □

Whether the interpreter terminates depends on the program, and since PALP is a Turing complete language, termination is undecidable, and we can refer to general termination proof methods that are based on sufficient conditions.

5.5 Other Variants of the PALP Query Interpreter

The principles for calculation of conditional probabilities and for best-first search described for the propositional case in sections 4.2 and 4.3 can be incorporated into the general PALP query interpreter described here with no problems, so we omit the details.

Any query which terminates correctly for a given PALP in the all-explanations version will also terminate correctly with the best-first version. Some programs may terminate with best-first, giving a best solution, but loop with all-solutions. This may happen when the program has a loop in a branch with lower probability, or if it has an infinite number of explanations.

There is a small blemish in the best-first interpreter as it may emit non-minimal explanations containing non-ground abducibles. This comes from the fact that the search is controlled by probabilities, which means that

```
queue_explain([],[a(_),b], 0.5)          (48)
```

may be selected before

```
queue_explain([Rest],[b],0.5);          (49)
```

and it may be the case the *Rest* succeeds later without referring to other abducibles (see also proposition 3 with remarks, above).

The remedy is to hold back final explanations with non-ground abducibles, as in (48), until there are no subsuming explanations with the same probability as in (49). In the example, this means that (48) must wait until (49) has transformed into $c =$ `queue_explain([],[b],0.5)`. Then c would be selected and printed out before (48), and (48) then immediately eliminated by the subsumption removal rule (2nd rule of (32) above).

Another efficient, but admittedly *ad-hoc* approach, is to fake a probability of 0.999 to non-ground abducibles instead of the correct value 1. This may work correctly in all but extreme cases.

6 Optimizations and Extensions

The architecture of the query interpreters described above provide a flexibility to plug in different optimizations and extensions, of which we consider some examples here.

6.1 Optimization *à la* Dijkstra's Shortest Path Algorithm

We suggest here an optimization of the best-first query interpreters inspired by Dijkstra's shortest path algorithm [21]. Whenever we have two or more processes with the same remaining subgoal (e.g., for finding a path from the same intermediate node to the terminal node in the shortest path example), we keep only the best one; in CHR:

```
queue_explain([G],E1,P1) \ queue_explain([G],E2,P2) <=>
prority_less_than(P2,P1) | true.          (50)
```

This will suppress the partial execution of some branches which are deemed not to become best in the end.

Notice that we indicated the rest query by a pattern that matches only queries with a single atom, which means that the rule is quickly bypassed for any query with two or more atoms. We could in principle have used a variable that matches any query, but this would lead to slower tests for matching of the two queries (and which likely fails in most cases).

If, furthermore, an analysis of the PALP under consideration tells which predicate(s) that may appear in singleton queries, we can make the pattern even more specific. An example of this optimization is given in section 7.2 below.

6.2 Optimizing Integrity Checks by Simpification

We mention also the possibility of applying simplified integrity constraints in specialized rules for each abducibles predicate. Simplification was suggested by [33] for database integrity checking; an unfolding of the theoretical foundations and a powerful method is given by [18]. The overall idea is to assume the database (here the current explanation) be consistent before an update, and based on that knowledge, to construct for each possible update a specialized check that considers only the part of the database which may interfere with the update. A typical speed up by this technique is an order of magnitude or more, when compared with a full check.

Integrity checking in our interpreters shown so far are hidden in the `extend` and `recalculate` auxiliary predicates, which do not take the actual update into account. Consider, as an example, the integrity constraint

```
⊥:- a(X), b(X).                                                        (51)
```

Without any special indexing techniques, this needs quadratic time measured in the size of the explanation E being checked, e.g., by a combination of two calls to `member`, `member(a(X),`E`)`, `member(b(X),`E`)`. If we know that explanation E is consistent, we can obtain by simplification for update `a(Y)` the linear check `member(b(Y),`E`)`.

We may now replace the generic rule for handling abducibles by specialized ones for each abducible predicate, e.g., as follows.

```
explain([a(X)|G],E,P) <=>
    (member(b(X),E) -> true   % vanish
     ;                                                                 (52)
     insert(a(X),E,E1), P1 is P*0.9,
     explain(Q,E1,P1)).
```

This principle can be further extended with specialized treatment for PALP clauses with more that one abducible in the body.

6.3 A Note on Negation

A limited form of explicit negation of abducibles can be implemented through integrity constraints. When `a/1` is an abducible predicate, we may let `not_a/1` stand for the negation of `a/1` and define the intended semantics by the integrity constraint $\bot$`:- a(X), not_a(X).`

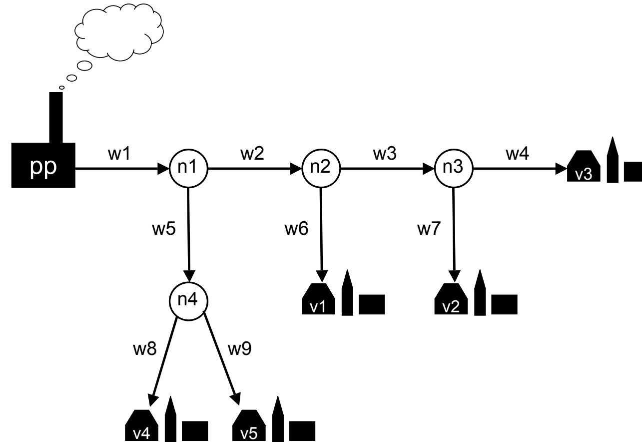
While this may be practical for many applications, we lack support for the other axiom for negation, namely `a(X)` $\vee$ `not_a(X).` We cannot handle this currently, as integrity checking becomes considerably more complicated. The extra axiom will imply that arbitrary logic programs can be encoded in the integrity constraints; the check, then, amounts to testing satisfiability of such programs, for which we have no straightforward embedding in CHR. See section 8.2 below which gives a suggestion for a more satisfactory treatment of negation.

We notice that the approach of [35], described in more detail in section 8.1 below, to probabilistic abduction includes negation with support of both axioms, but excludes integrity constraints and require any negated call to an abducible or defined predicate to be ground.

7 Program Examples

7.1 A Standard Diagnosis Case

We consider a power supply network which has one power plant `pp`, a number of directed wires $\mathtt{w}_i$ and connecting nodes $\mathtt{n}_i$, which may lead electricity to a collection of villages $\mathtt{v}_i$. The overall structure is as follows.



Probabilistic abduction will be used to predict to most likely damages in the network given observations about which villages have electricity and which have not. As abducibles, we use `up/1` and `down/1` which apply to the power plant and the wires (for simplicity, the connecting nodes are assumed always to work).

The network structure is represented by the following facts.

```
edge(w1, pp, n1).   edge(w4, n3, v3).   edge(w7, n3, v2).
edge(w2, n1, n2).   edge(w5, n1, n4).   edge(w8, n4, v4).        (53)
edge(w3, n2, n3).   edge(w6, n2, v1).   edge(w9, n4, v5).
```

The fact that a given point in the network has electricity, is described as follows.

```
haspower(pp):- up(pp).                                           (54)
haspower(N2):- edge(W,N1,N2), up(W), haspower(N1).
```

As no negation is supported, the program includes also clauses that simulate the negation of `haspower`.

```
hasnopower(pp):- down(pp).
hasnopower(N2):- edge(W,_,N2), down(W).                          (55)
hasnopower(N2):- edge(_,N1,N2), hasnopower(N1).
```

To express that `up/1` and `down/1` are each other's negation, we introduce an integrity constraint, and define probabilities that sum to one.

```
abducible(up(_), 0.9).
abducible(down(_), 0.1).                                         (56)
⊥:- up(X), down(X).
```

The predicate definitions are compiled in CHR as explained above; we show here the one for the `haspower` predicate.

```
step_explain( [haspower(N)|G], E, P) <=>
    rename([haspower(N)|G], [haspower(Nr1)|Gr1]),
    (Nr1=pp -> queue_explain([up(pp)|Gr1], E, P) ; true),        (57)
    queue_explain([edge(W2,N12,N),up(W2),haspower(N12)|G],E,P),
    select_best.
```

The implementation of the `extend` auxiliary (which is used when a new abducible is encountered) includes the checking of the integrity constraint. The following excerpt of a screen dialogue shows how the observation that no village have

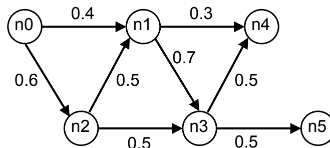
electricity is explained by the interpreter.

```
| ?- explain([hasnopower(v1), hasnopower(v2),
     hasnopower(v3), hasnopower(v4), hasnopower(v5)]).
Best solution: [down(w1)]
Prob=0.1
Another solution? y
Best solution: [down(pp)]
Prob=0.1
Another solution? y
Best solution: [down(w2),down(w5)]
Prob=0.01
Another solution? y
Best solution: [down(w3),down(w5),down(w6)]
Prob=0.001
Another solution? y
Best solution: [down(w2),down(w8),down(w9)]
Prob=0.001
Another solution?
⋮
```
(58)

It appears that the two intuitively most reasonable hypotheses, namely that the power plant or the single wire connecting it with the rest of the network is down, are generated as the first ones with highest probability. Then follow combinations with lower and lower probability of different wires being down. The original output indicated insignificant rounding errors in the calculated probabilities which have been retouched away above.

7.2 Most Probable Path with Dijkstra Optimization

Here we illustrate both the optimization described in section 6.1 above for a best-first interpreter and an extended syntax for declaration of abducibles. We consider the problem of finding most probable paths through a network such as the following.



The figures for the outgoing edge of a node indicate the probability for choosing a particular edge from that node. We could in principle declare one nullary abducible predicate for each edge, but to facilitate writing the PALP, we use one common predicate `select(`n`,`m`)` describing the event that the indicated edge is chosen. We declare it as follows, extending the syntax of definition 1 above.

```
abducible(select(n0,n1), 0.4).
abducible(select(n0,n2), 0.6).                                    (59)
etc.
```

I.e., we have several declarations for the same abducible predicate, specifying different probabilities for different arguments. The intuitively correct semantics is preserved provided that no two declared abducible atoms can unify. The implementation needs one single adjustment so that the call to an abducible predicate, say `select(n0,X)`, launches a new branch for each possible choice of declaration with which it unifies; this is done analogously to the way that defined predicates are handled (section 5.2, above). The path program can be implemented as follows, using the "generic" abducible predicate.

```
path(N1,N3):- select(N1,N2), path(N2,N3).                         (60)
path(N,N).
```

We may add integrity constraints of the form $\bot$`:- select(`n`,`x`), select(`n`,`y`)`, for all cases of $n \rightarrow x$ and $n \rightarrow y$ being different differet edges going out from n, but due the best-first search, they are in fact not necessary.

This program is translated into a best-first query interpreter in CHR as decribed above, and we add the following rule in order to prune any initial path segment, which is less optimal than another such segment ending in the same node.

```
queue_explain([path(N,M)],_,P1)
    \ queue_explain([path(N,M)],E,P2)                              (61)
<=> P2 < P1 | true.
```

The query

```
?- explain(path(n0,n4)).                                          (62)
```

provides one answer, namely

```
[select(n0,n2),select(n2,n3),select(n3,n4)], Prob=0.15.           (63)
```

No more answers are produced as rule (61) has removed all segments that could lead to less optimal paths through the graph. A test print indicates that the following constraints have been deleted by this rule.

```
queue_explain(path(n1,n4),[select(n0,n2),select(n2,n1)],0.3)
queue_explain(path(n3,n4),[select(n0,n1),select(n1,n3)],0.28) (64)
queue_explain(path(n4,n4),[select(n0,n1),select(n1,n4)],0.12)
```

8 Conclusion

We have defined a class of Probabilistic Abductive Logic Programs and described implementations in terms of a systematic transformation into CHR rules. This framework differs from other approaches to probabilistic logic programming (that we are aware of) by having both interaction with external constraint solvers and integrity constraints. We support no general negation in abductive logic programs, as is done in several methods for non-probabilistic abduction, and we have argued that (at least our approach to) the probabilistic semantics is difficult to adapt to negation; we have, however, indicated how a simplified version of explicit negation can be implemented with integrity constraints.

8.1 Related Work

Abduction in logic programming without probabilities has attracted a lot of attention, and several algorithms, including metainterpreters written in Prolog have been made; see [28, 20] for overview and references. We may emphasize an early work by Console *et al* [19] 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 [27], which inspired several implemented systems. Ignoring the probabilistic part of our own interpreters, they show similarity with the principle of [19] in the sense that we map abductive programs into CHR, which is a purely deductive paradigm; as shown in lemmas 4, 5, the execution state represents at any time the semantics given by the initial query and any transformation made by a CHR rules can be explained from and respects the program completion.

Abduction without probabilities has been approached using CHR, initially by [1] translating abductive logic programs into the dialect called $\text{CHR}^\vee$ [2] that features disjunctions in rule bodies. In that approach, abducibles are represented directly as CHR constraints and integrity constraints as CHR rules, and predicate definitions are translated into $\text{CHR}^\vee$ with a disjunct for each clause. In later work [16], this principle has been modified by representing the clauses of an abductive logic program directly as their Prolog equivalents, leading to a very efficient implementation of abduction with no interpretational overhead; [13] provides an overview of this direction and extends with methods for interaction with arbitrary external constraint solvers, similarly to what we have explained in the present paper in a probabilistic version. These implementations could in principles be adapted to top-down (but not best-first) abduction, simply by calculating the probability for each generated answer when printing it out. However, integrity constraints would here need to be limited to the sort we use in the present paper, as an instance of the more general pattern such as `a,b ==> c` indicates a probabilistic dependency, which our semantics is not prepared for; an analogous phenomenon was discussed in 6.3 in relation to negation.

In [16], it is also shown how so-called assumptions can be implemented in a similar way with CHR; assumptions are like abducibles, but with explicit cre-

ation, application (perhaps being consumed) and scope rules; [15, 11, 17, 16] show linguistic applications of logical grammars (as DCGs or bottom-up parsing with CHR) extended with abduction using CHR. A notion of Global Abduction [40, 41], allowing a sort of destructive non-monotonic updates and interaction between different processes (or agents) have been implemented in CHR by [12] using the constraint store as a process pool, as in the present paper.

In [10], a reversal implementation of the proof predicate `demo(`p`,`q`)`, meaning that query q succeeds in program p, is described and implemented using CHR for the primitive operations within the metainterpreter that defines the proof predicate. Reversibility means that it can fill in missing parts of the program argument in order to make specified queries provable, and thus it can also perform abduction, although no notion of minimality is supported.

Recent approaches to abductive logic programming, e.g., [29, 23, 4], have studied the interaction with externally defined constraint solvers, but implementations tend to be specialized to specific constraint solvers. SCIFF [4] is an approach to abductive logic programming which includes negation, integrity constraints, external constraints solvers, and other specialized facilities; the existence of an implementation made with CHR is indicated in [3], but no details are given which allow for a comparison.

Probabilistic versions of abductive logic programming have not been studied nearly to the same extent. We can refer to the work by D. Poole [34] considering probabilistic abduction for a class of Probabilistic Horn Abduction Theories; this is later [35] generalized into Independent Choice Logic. Abducible predicates are grouped by so-called *disjoint declarations* of the form `disjoint([`a_1`:`p_1`, ..., `a_n`:`p_n`])`. The intension is that common instances of a_i, a_j, $i \neq j$ cannot coexist in the same explanation, corresponding to integrity constraints $\bot$`:-`a_i, a_j for all $i \neq j$; other integrity constraints are not possible. Probabilities are given by $P(a'_i) = p_i$ for a ground instance a'_i of a_i, and it holds that $p_1 + \cdots + p_n = 1$. The framework does not assign probabilities to non-ground abducibles. A metainterpreter written in Prolog is described in [34], which works best-first using a probability ranked priority queue analogous to what we have described (however, with a more complicated way of attaching probabilities to items in the queue). In [35], the appoach is extended for negation as failure of ground goals G, presupposing that the set of all minimal explanations $\{E_i\}_{i \in I}$ is finite and each of those finite and always ground. In such a case, explanations for the negation of G can be found by regrouping of negated elements of the E_i explanations; this excludes best-first search as the interpreter needs to keep track of all explanations.

The PRISM system [38] is a powerful reasoning system, which is based on logic programming extended with multivalued random variables that work slightly differently from abducible predicates as descibed in the present paper, but it is straightforward to rewrite an abductive logic program into a PRISM program. PRISM has no support for integrity constraints or interface to external constraint solvers. PRISM includes a variety of top-level predicates which can generate abductive explanations, including finding the best ones using a general-

ized viterbi algorithm. Another central feature of PRISM is its machine learning capabilities, which means that in can learn probabilities from training data.

Reasoning in Bayesian networks can also be considered an instance of probabilistic abduction, but we will refrain from giving detailed references, since the knowledge representations are different. Bayesian networks are easily embedded in abductive logic programming and can be simulated in our system as well as [34, 38]. One of the advantages of Bayesian networks is that there exist very efficient implementations which can find approximative solution for huge networks.

Logic programs with associated probability distribution have been used elsewhere, including for inductive logic programming, but the issue of abduction does not seem to have been addressed; e.g., [32, 36].

Probabilistic Constraint Handling Rules are introduced by [26]; probabilities are assigned to each rule of a program for it to apply and it is defined by a an operational semantics and implemented by a transformation into CHR; [30] describes user-defined priorities for CHR.

There is an inherent similarity between answer set programming (ASP) and abductive reasoning with integrity, which has been noted by many authors; [6] describes an extension of ASP with probabilities which, thus, is capable of doing probabilistic abductive reasoning (no implementation is reported, though). However, this framework excludes programs that exhibit the property illustrated in example 7, that the probability of abducibles considering consistent states only is different from the probability defined by the programmer; this means that many probabilistic abductive programs with integrity constraints are not covered. By nature, ASP programs can only produce ground abductive explanations.

8.2 Perspectives and Future Work

Obvious applications of our framework seem to be diagnosis and stochastic language processing. Relatively efficient methods exist for stochastic context-free grammars already, but we may approach property grammars [8, 7] which are a formalism based entirely on constraint satisfaction rather than tree structure; by nature, these grammars have a very high degree of ambiguity so a probabiistic approach using best-first search may be relevant.

Probabilistic extensions of Global Abduction (see related work section above) or similar frameworks may be relevant to apply for applications monitoring and interacting with the real world. It seems also possible to extend the probabilistic best-first search strategy to take into account changing probabilities, e.g., produced by a learning agent or an agent monitoring specific subsystems by means of, say, a Bayesian network.

The present approach can be immediately generalized for arbitrary monotonic priority functions, e.g., represent some object function to be optimized or adjusted probabilities; in computational linguistics in may be relevant to use adjusted probabilities for partial explanations according to the length of the text segment they represent. See [14] for an initial publication on this approach; it is also relevant to compare with [30] that considers CHR with rule priorities.

In order to extend the approach with negation and maintain a relatively good efficiency, the principle of compiling a logic program into another one that expresses its negation is under consideration; see [5, 39] for such methods. The example of section 7.1 showed a trivial and manually produced example of such a translation.

Finally, we mention that our implementation principle, transforming PALPs systematically into CHR, can be embedded in a compiler, so that PALPs can be written in Prolog source files and compiled automatically into CHR. Prolog's metaprogramming facilities including the so-called term expansion facilities, see, e.g., [43], make the implementation of such a compiler a minor task; [11, 16] explain systems based on CHR implemented in this way.

A Proofs for Important Properties

Proof (lemma 2). Let $\{E_1, \ldots, E_n\}$ be a complete set of minimal explanation for Q in a PALP Π and E an arbitrary explanation for Q. Since $\Pi \cup \exists E \models [\![Q]\!]$ and $\Pi \models [\![Q]\!] \leftrightarrow \exists E_1 \vee \cdots \vee \exists E_n$, both by definition, we have that $\Pi \models \exists E \rightarrow \exists E_1 \vee \cdots \vee \exists E_n$ and thus $\models \exists E \rightarrow \exists E_1 \vee \cdots \vee \exists E_n$. Since $E, E_1, \ldots, E_2$ are conjunctions of atoms, this means that $\models \exists E \rightarrow \exists E_i$ for some E_i, $1 \leq i \leq n$ which the same as E_i subsumes E.

From this part of the lemma, the uniqueness of complete sets of minimal explanations follows immediately. □

Proof (lemma 3). It is sufficient to show that every E_i is a minimal explanation. Clearly E_i is an explanation, and according to lemma 1 there is a minimal explanation E'_i with $E'_i \subseteq E_i$. As in the proof of lemma 2, we find that $\models E'_i \rightarrow E_1 \vee \cdots \vee E_n$: By assumption we cannot have $\models E'_i \rightarrow E_j$ for $i \neq j$, which means that $\models E'_i \rightarrow E_i$. In other words $E_i \subseteq E'_i$ and thus $E_i = E'_i$. □

Proof (lemma 4). The initial constraint store {`explain(`Q`,[],1)`} satisfies the property. It is straightforward to verify that each of the CHR rules (21,22,23) preserves the property, so it follows by induction that it holds for any subsequent constraint store. □

Proof (theorem 1). Termination is guaranteed as a PPALP has no recursion, and since each step performed by CHR rules (21,22) introduces new `explain/3` constraints, each of which represents a step in an SLD derivation in $\Pi \cup A$ where A is the set of all abducibles.

From lemma 4 and the fact that rule (23) removes any `explain(_,`E_i`,_)` constraint for which there is another `explain([],`E_j`,_)` with $E_j \subseteq E_i$, $i \neq j$, it can be seen the final constraint store is of the form

$$\texttt{explain([],}E_1\texttt{,}p_1\texttt{)}, \ldots, \ \texttt{explain([],}E_n\texttt{,}p_n\texttt{)} \tag{65}$$

where $E_j \not\subseteq E_i$ for all $i \neq j$. The theorem follows now from lemmas 3 and 4. □

Proof (lemma 5). As in the proof of lemma 4, we notice that the initial constraint store satisfies the property and that each possible derivation step preserves the property. It should be noticed that rules (39,21,40) in some cases suppress constraints explain(Q,E,p) for which $\Pi \not\models [\![Q, E]\!]$. □

Proof (theorem 2). The arguments are identical to those in the proof of theorem 1 except that we refer to lemma 5 instead of lemma 4. □

B 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.

B.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

% normalize/2 not relevant 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).
```
(66)

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).
```
(67)

B.2 For Nonground Abducibles

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.

% rename/2, as above

% inconsistent/2, as above

% subset/2 as above
```

The predicate `normalize_final(`$E1$`,`$E2$`)` is in its present version defined in a way so it tries out all possible subsets, and selects as E_2 a smallest one which is equivalent to E_1; *in the final version of this paper, we may provide a more efficient implementation.*

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. Slim Abdennadher and Heribert Schütz. CHR$^\vee$: A flexible query language. In Troels Andreasen, Henning Christiansen, and Henrik Legind Larsen, editors, *FQAS*, volume 1495 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1998.
3. Marco Alberti, Federico Chesani, Marco Gavanelli, and Evelina Lamma. The CHR-based implementation of a system for generation and confirmation of hypotheses. In Armin Wolf, Thom W. Frühwirth, and Marc Meister, editors, *W(C)LP*, volume 2005-01 of *Ulmer Informatik-Berichte*, pages 111–122. Universität Ulm, Germany, 2005.
4. Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Verifiable agent interaction in abductive logic programming: the SCIFF framework. *ACM Transactions on Computational Logic*, 9(4), 2008. To appear.
5. José Júlio Alferes, Luís Moniz Pereira, and Terrance Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *Theory and Practice of Logic Programming*, 4(4):383–428, 2004.
6. Chitta Baral, Michael Gelfond, and J. Nelson Rushton. Probabilistic reasoning with answer sets. In Vladimir Lifschitz and Ilkka Niemelä, editors, *LPNMR*, volume 2923 of *Lecture Notes in Computer Science*, pages 21–33. Springer, 2004.
7. Philippe Blache. Property grammars: A fully constraint-based theory. In Henning Christiansen, Peter Rossen Skadhauge, and Jørgen Villadsen, editors, *CSLP*, volume 3438 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2004.
8. Philippe Blache and Jean-Marie Balfourier. Property grammars: a flexible constraint-based approach to parsing. In *IWPT*. Tsinghua University Press, 2001.

9. Christian Holzbaur. OFAI clp(q,r) Manual, Edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.
10. Henning Christiansen. Automated reasoning with a constraint-based metainterpreter. *Journal of Logic Programming*, 37(1-3):213–254, 1998.
11. Henning Christiansen. CHR Grammars. *Int'l Journal on Theory and Practice of Logic Programming*, 5(4-5):467–501, 2005.
12. 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.
13. Henning Christiansen. Executable specifications for hypotheses-based reasoning with Prolog and Constraint Handling Rules. *Journal of Applied Logic*, 2008. to appear.
14. Henning Christiansen. Prioritized abduction with CHR. In Tom Schrijvers, Frank Raiser, and Thom Frühwirth, editors, *CHR 2008, The 5th Workshop on Constraint Handling Rules (proceedings); RISC-Linz Report Series No. 08-10*, pages 159–173, 2008.
15. Henning Christiansen and Verónica Dahl. Logic grammars for diagnosis and repair. *International Journal on Artificial Intelligence Tools*, 12(3):227–248, 2003.
16. 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.
17. Henning Christiansen and Verónica 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.
18. Henning Christiansen and Davide Martinenghi. On simplification of database integrity constraints. *Fundamenta Informatica*, 71(4):371–417, 2006.
19. 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.
20. 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.
21. Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(4):269–271, 1959.
22. Gregory J. Duck, Peter J. Stuckey, Maria J. García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2004.
23. 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.
24. 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.

25. 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.
26. Thom W. Frühwirth, Alessandra Di Pierro, and Herbert Wiklicky. Probabilistic constraint handling rules. *Electronic Notes in Theoretical Computer Science*, 76, 2002.
27. Tzee Ho Fung and Robert A. Kowalski. The iff proof procedure for abductive logic programming. *Journal of Logic Programmming*, 33(2):151–165, 1997.
28. 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.
29. A.C. Kakas, A. Michael, and C. Mourlas. ACLP: Abductive Constraint Logic Programming. *Journal of Logic Programming*, 44:129–177, 2000.
30. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for chr. In Michael Leuschel and Andreas Podelski, editors, *PPDP*, pages 25–36. ACM, 2007.
31. John W. Lloyd. *Foundations of logic programming; Second, extended edition.* Springer-Verlag, 1987.
32. Stephen Muggleton. Stochastic logic programs. In Luc de Raedt, editor, *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, 1996.
33. Jean-Marie Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.
34. David Poole. Logic programming, abduction and probability - a top-down anytime algorithm for estimating prior and posterior probabilities. *New Generation Computing*, 11(3):377–400, 1993.
35. David Poole. Abducing through negation as failure: stable models within the independent choice logic. *Journal of Logic Programming*, 44(1-3):5–35, 2000.
36. Stefan Reitzler. *Probabilistic Constraint Logic Programming.* PhD thesis, 1998. Appearing as AIMS, Arbeitspapiere des Instituts für Maschinelle Sprachverarbeitung, Lehrstuhl für Theoretische Computerlinguistic, Universität Stuttgart, Vol. 5, No. 1, 1999.
37. John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
38. Taisuke Sato and Yoshitaka Kameya. Prism: A language for symbolic-statistical modeling. In *IJCAI*, pages 1330–1339, 1997.
39. Taisuke Sato and Hisao Tamaki. First order compiler: A deterministic logic program synthesis algorithm. *Journal of Symbolic Computation*, 8(6):605–627, 1989.
40. Ken Satoh. "All's well that ends well" - a proposal of global abduction. In James P. Delgrande and Torsten Schaub, editors, *NMR*, pages 360–367, 2004.
41. Ken Satoh. An application of global abduction to an information agent which modifies a plan upon failure - preliminary report. In João Alexandre Leite and Paolo Torroni, editors, *CLIMA V*, volume 3487 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2004.
42. 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.
43. 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.