Assumptions and Abduction in Prolog

Henning Christiansen[1] and Veronica Dahl[2]

[1] Roskilde University, Computer Science Dept.
P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: henning@ruc.dk

[2] Dept. of Computer Science
Simon Fraser University
Burnaby, B.C., Canada
E-mail: veronica@cs.sfu.ca

Abstract. Abduction is agreed upon as a powerful technique in logic programming but its actual use in practice appears to be rather limited since most available systems are research prototypes implemented using inefficient metaprogramming techniques. Assumptive logic programming is related to abduction but provides explicit creation and consumption of hypotheses plus scoping principles inspired by linear logic. We show how a class of abductive logic programs (and assumptive logic programs) can be executed directly in Prolog using a trivial extension for abducible predicates (assumptions) written in Constraint Handling Rules; this version of assumptions extends earlier approaches with integrity constraints. The motivation behind the present work is to show that abduction and assumptions can be integrated with traditional Prolog programs without any significant slow-down in execution speed or other burdens for the programmer.

1 Introduction

Assumption-based reasoning in general, or hypothetical reasoning is defined by [17] as a logic system in which a set of facts and a set of possible hypotheses are given. Its instances can be assumed if they are consistent with the facts. Both abduction (the unsound but useful assumption of B given A and given that B implies A) and linear and intuitionistic logic inspired assumptions [20] fall into that general category. Their formalization within, respectively, Abductive Logic Programming [14] and Assumptive Logic Programming [9] refines this general notion by for instance requiring in the first case consistency with a special type of facts: integrity constraints. Both allow us to move beyond the limits of classical logic to explore "possible cause" and "what-if" scenarios. They have proved useful for diagnosis, recognition, difficult human language processing problems, and many other applications. However in practice, abduction in particular has not been used to its full potential owing to implementation indirections. Assumptions can be more efficiently implemented through continuation based processors

such as BinProlog, but there is no Prolog in existence which efficiently provides both capabilities at the same time.

In this paper we show that, rather than introducing an extensive implementation apparatus, we can achieve a very direct while efficient implementation in Prolog as a programming language, through simply extending it with a few lines of CHR code (Constraint Handling Rules [12]) to handle the abducibles or assumptions. This provides an optimal combination, in which programs can be written and executed directly, with only a small extra overhead involved when needed. The price paid is a limited use of negation. Yet even with this restriction, many useful examples are made possible.

Another way of viewing our present contribution is as an extension of earlier approaches to assumptive logic programming with integrity constraints and integrating it with abduction into a paradigm that we may call A^2LP.

Our implementations use SICStus Prolog [18] and its CHR library; we refer to the proper sections of the referenced manual for a detailed description of the facilities that we use; in addition we use an undocumented feature for improved efficiency which we will explain.

Overview

Section 2 explains how a class of Abductive Logic Programs is represented in our methodology, a small example is given and the limitations are pointed out. Assumptive Logic Programs are treated in a similar way in section 3. In section 4, we show how the two paradigms become useful for language processing when combined with Definite Clause Grammars. The final section 5 gives conclusion and discussion of related work.

2 Abduction

An abductive logic program [14] is usually specified as a triplet $\langle \mathcal{P}, \mathcal{A}, \mathcal{C} \rangle$ where $\mathcal{P}$ is a logic program, $\mathcal{A}$ a set of *abducible* predicates that do not occur in the head of any clause of $\mathcal{P}$, and $\mathcal{C}$ a set of integrity constraints assumed to be consistent.

Assume additionally that $\mathcal{P}$ and $\mathcal{C}$ can refer to a set of *built-in* predicates that have a fixed meaning identified as a theory $\mathcal{B}$; a predicate in $\mathcal{P}$ that is neither abducible nor built-in is called *defined*. A typical built-in is `dif/2` stating that two terms are syntactically different and for which SICStus Prolog provides a correct implementation. We assume for simplicity in the following that $\mathcal{C}$ refers to abducible and built-in predicates only. In the following we use Prolog notation for logic programs and CHR propagation rules for integrity constraints; we use the implementation syntax indicating each abducible atom by a prefix exclamation mark (not to be confused with Prolog's cut).

In the context of an abductive logic program $\langle \mathcal{P}, \mathcal{A}, \mathcal{C} \rangle$, we define for pairs of sets of abducibles and built-in atoms $\langle A, B \rangle$, a *consistent ground instance* to be a common ground instance $\langle A', B' \rangle$ of $\langle A, B \rangle$ so that

– $\mathcal{B} \models B'$ (the instance of built-ins is satisfied)
– $A' \models \mathcal{C}$ (the instance of abducibles respects the integrity constraints)

Example 1. Consider abducible predicates `!a/1` and `!b/1` and an abductive program that contains the integrity constraint `!a(X), !b(X) ==> fail.` Let S be the pair of sets $\langle\{$`!a(X)`, `!b(Y)`$\}, \{$`dif(X,7)`$\}\rangle$. Then a consistent ground instantiation of S is obtained by substitution `X=1`,`Y=2`, whereas `X=1`,`Y=1` or `X=7`,`Y=1` do not give rise to consistent ground instantiations.

For simplicity and without loss of generality, we consider only ground queries; an *abductive answer* to a query Q is a pair of finite sets of abducible and of built-ins atoms $\langle A, B\rangle$ such that

– $\langle A, B\rangle$ has at least one consistent ground instance $\langle A', B'\rangle$,
– for any such $\langle A', B'\rangle$, we have $\mathcal{P} \cup A' \models Q$.

We now turn our attention to our implementation and define a *negation-free* abductive logic program as one which has no application of negation, whose abductive predicates are distinguished by prefix exclamation marks (so, e.g., `p` and `!p` refer to different predicates), and whose integrity constraints are written as CHR propagation rules whose head atoms are abducibles and whose body atoms are abducibles or built-ins (or possibly `fail`).

Example 2. The following is a standard example of integrity constraints that are written in the syntax of CHR's propagation rule.

```
!in_buenos_aires(X), !revolution ==> !school_cancelled(X).
!in_vancouver(X), !snow ==> !school_cancelled(X).
!revolution, !snow ==> fail.
```

Example 3. The following integrity constraint concerns a predicate for marriages `m(`*husband*`,` *wife*`)`; it uses the built-in predicate for unification to ensure that any man can only have one wife.

```
!m(X,Y), !m(X,Z) ==> Y=Z
```

The implementation in Prolog with CHR is simple: Abducibles are viewed as constraints in the sense of CHR, the logic program is executed by the Prolog system and whenever an abducible is called it is added automatically by CHR to the constraint store and CHR will activate integrity constraints whenever relevant. The complete implementation in SICStus Prolog is provided by including the following lines in the start of a the program file.[3]

```
:- use_module(library(chr)).
:- op(500,fx,!).
handler abduction.
constraints ! /1.
```

[3] The prefix exclamation mark is used as a "generic" abducible predicate. This is mainly for simplicity of the presentation, and it is obvious that an additional speed-up can be gained by compiling a specialized version of the machinery below for each individual abducible predicate.

Example 4. Consider the integrity constraints of example 2 and a Prolog program which, as part of solving its task, calls abducibles at certain points. In case `!revolution` is entered as the first abducible, it is added to the constraint store but no integrity constraint is activated. If subsequently `!in_buenos_aires(P)` is called, it is added to the constraint store and CHR figures out to call the first integrity constraint, thus introducing the additional abducible `!school_cancelled(P)`; notice that the method handles correctly abducibles that are parameterized by variables without any extra machinery (this was a problem in some early abduction algorithms). If, furthermore, abducible `!snow` is called, this triggers the third integrity constraint producing a failure which in turn forces the Prolog program to backtrack. In example 3, the sequence of calls `!m(peter,A)`, `!m(peter,mary)` will trigger the integrity constraint so that variable `A` is unified with `mary`.

The correctness of this implementation of negation free abductive programs is inherited from the correctness properties of the underlying Prolog plus CHR systems. For any program without occurs-check problems, the implementation produces correct abductive answers as defined above; if the program (including integrity constraints) does not loop, we also have that the total set of answers is complete.

It is interesting to notice that the approach can interact with an arbitrary constraint solver by considering its constraints as built-ins (applied in bodies of clauses and integrity constraints). Possible soundness and completeness of such a combination will mirror the properties of the applied constraint solver.

Minimality

It is often required that an abductive answer be minimal measured in the number of abduced literals (or, alternatively, in a subset relation or subsumption ordering). Most published abduction algorithms include a device that tries to unify a new abducible with one already produced during the construction of a proof, and tries out different alternatives under backtracking. This does not guarantee minimality if, say, the chosen proof needs `!a` and `!b` but another proof may need only `!a`. Minimal answers can be selected by a post-processing of all answers found in this way.

There are, however, examples in which the minimal explanations produced by compaction of nonminimal ones by unification seem overconstrained. If, say, my wallet is stolen in one city and my car in another city, it is groundless to assume the thief is the same unless, of course, there is special evidence for the fact (which may be imposed by an integrity constraint).

Nevertheless, we can easily provide a device that dynamically tries to unify a new abducible with an existing one. Add the following rule to the program (the empty guard appearing as "`true |`" can be ignored but is needed due to a syntactic peculiarity in SICStus Prolog's version of CHR).

```
!A , !B  ==> true | (A=B ; dif(A,B)).
```

The rule implements the compaction principle correctly but has a few disadvantages. If `A` and `B` are identical at the time of the call, it results in keeping the two identical abducibles in the state (which may cause later applications to repeat the same work twice) plus setting up a useless choice point. If `A` and `B` are nonunifiable at the time of the call, the execution of the body will be waste of time. The following rule provides an optimal behaviour:

```
!A , !B#X  ==> true |
    (A==B -> remove_constraint(X)
     ;   ?=(A,B) -> true
     ;  (remove_constraint(X), A=B ; dif(A,B)))
  pragma passive(X).
```

The `pragma passive` annotation ensures that the rule is not applied twice when a new abducible atom shows up. The use of the "`#X`" notation to provide a dynamic handle to the matched CHR constraints (`B`) is undocumented but serves a good purpose here. In the last case, when `A` and `B` are unifiable but not identical so that both branches are possible, we remove one of the constraints before the unification in order to avoid the duplicate, and under backtracking the constraint is put back into the store after the unification has been undone.

Negation

A limited version of explicit negation can be implemented by means of an integrity constraint. Extend the code shown so far with the following:

```
:- op(499,fx,[not]).
!A, !not A ==> fail.
```

Now a program clause as well as integrity constraints can refer to negated abducibles, and the above CHR rule will prevent the creation of an abducible and its negation.

Example 5. We can extend the integrity constraints in example 2 with the following that uses negation to state that revolutions are not possible in Vancouver.

```
!in_vancouver(X) ==> !not revolution.
```

If the controlling Prolog program has generated abducible `!revolution`, and then `!in_vancouver(v)`, firstly the integrity constraint above produces `!not revolution` which immediately forces the generic rule for explicit negation to produce a failure that corresponds to the inherent inconsistency. Notice that the only code involved in the execution is some Prolog program (not shown) and the CHR rules shown. The CHR system (with whatever indexing techniques it may use under the surface) guarantees an optimal handling of the abducibles and integrity constraints.

Although useful for many applications, this implementation covers only one part of negation "you cannot have P and $\neg P$ at the same time"; the condition saying that "either you have P or $\neg P$" cannot be expressed in a straightforward way.

It is possible in many cases to achieve this effect by a transformation of integrity constraints.

Example 6. Consider an abductive program with the following integrity constraints (and the "`!A, !not A ==> fail`" above).

```
!a, !b ==> fail.
!not b, c ==> fail.
```

Clearly the sequence `!a, !c` ought to result in failure, but our implementation explained so far will return the two abducibles. The desired effect can be obtained by a translation of the integrity constraints illustrated for the example as follows.

```
!a ==> true | (find_constraint(!not b) -> true ; !not b).
!b ==> true | (find_constraint(!not a) -> true ; !not a).
!not b ==> true | (find_constraint(!not c) -> true ; !not c)
c ==> (find_constraint(!b) -> true ; !b).
```

However, this translation principle explodes in numbers of special cases with more complicated integrity constraints and it is not clear that it can be extended to cover all cases with variabes. [*We will give more precise statements at this point in the final version of this paper*].

If a program clause includes a negated call that refers to abducibles directly or indirectly, we inherit the dubious semantics of Prolog. So with definition `p(X):-!a(X)`, a call `\+p(Z)` (where `Z` is a currently uninstantiated variable) may succeed in case the abduction of `!a(Z)` triggers a failure producing integrity constraint.

The problem is that Prolog's negation as failure removes any trace of what took place inside the call; a correct implementation should export the knowledge `!not a(Z)`. We currently have no suggestion for improvement at this point and it is also clear that a restriction to "safe negation" (delaying the call until it becomes ground) will still be problematic. However, a recent paper [15] provides a variant of constructive negation [4] that may be useful in some cases.

3 Assumptive logic programming

Assumptive logic programs [9] are logic programs augmented with a) linear, intuitionistic and timeless implications scoped over the current continuation, and b) implicit multiple accumulators, useful in particular to make the input and output strings invisible when our program describes a grammar (in which case we talk of Assumption Grammars [11]). Hidden accumulators allow us to disregard the input and output string arguments, as in DCGs, but with no preprocessing requirement. More precisely, we use the kind of linear implications called *affine* implications, in which assumptions can be consumed at most once, rather than exactly once as in linear logic.

We apply here a later and more homogeneous syntax for assumptions introduced in [7]; we do not consider accumulator and Assumption Grammars can be obtained applying the operators below in a DCG as we show in section 4.

`+h(a)`	Assert linear assumption h(a) for available for subsequent text. Linear means "can be used once".
`*h(a)`	Assert intuitionistic assumption for subsequent proof steps. Intuitionistic means "can be used any number of times"
`-h(X)`	Expectation: consume/apply existing assumption.
`=+h(a)`, `=*h(X)`, `=-h(X)`	As above but the time of assertion and application or consumption can be arbitrary
`expections_satisfied`	Tests whether all expectations has been met by an assumption; should be applied as the last thing in a query.

These operators are defined as constraints in CHR and can be called from the body of program rules; no sort of negation is possible. Integrity constraints can be written as any sort of CHR rules.

Assumption grammars have been used for natural language problems such as free word order, anaphora, coordination, and for knowledge based systems and internet applications. Assumptive logic programs are useful, among other things, for simulation of producer-consumer and resource allocation systems as illustrated by the following example.

Example 7. Consider a local area network which has a fixed number of printers, each characterized by its name its printing speed. This is represented by Prolog facts such as the following.

```
seconds_per_page(epsmark1993, 20).
seconds_per_page(lexon2000, 10).
seconds_per_page(pewhack2004, 2).
```

At any time, the status of each printer is represented by an assumption `+printer(` *name*`,` *ready-time*`)`. If *ready-time* is less than or equal to the current time it means that the printer is idle and can be give a job; if *ready-time* is greater that current time it means that the printer is currently occupied. The following Prolog program applies these to simulate a print scheduler that receives a list of print jobs and distributes it to the different printers (for simplicity we ignore arrival times for jobs); a list describing the job history is generated.

```
run([],_,[]).

run([(Id,Np) | Js], T, [printed(Id,P,FinAt)|Ds]):-
 -printer(P,ReadyAt),
 ReadyAt=<T,
 seconds_per_page(P,Spp),
```

```
    FinAt is T + Np*Spp,
    +printer(P, FinAt),
    run(Js,T,Ds).

  run(Js,T,Ds):-   % busy wait if no printer ready
    \+ ( -printer(_,ReadyAt), ReadyAt=<T),
    T1 is T+1, run(Js,T1,Ds).
```

The last clause employs Prolog's negation-as-failure to check assumptions without affecting the state. Before running a list of jobs, the printers must be initialized as in the following query and answers produced by our implementation described below.

```
 ?- +printer(epsmark1993,0), +printer(lexon2000,0),
                                        +printer(pewhack2004,0),
    run([(file1, 10),(file2,6),(file3,100), (file4,5)],0,H).
 H = [printed(file1,pewhack2004,20),printed(file2,lexon2000,60),
      printed(file3,epsmark1993,2000),
      printed(file4,pewhack2004,30)],
+(printer(lexon2000,60)),
+(printer(epsmark1993,2000)),
+(printer(pewhack2004,30)) ?
```

Notice that the final state also includes final status for the printers. We can illustrate the use of integrity constraints for assumption sketching an extension of the example. Assume all printers are covered by the same undersized electrical fuse that will melt down in case all three printers are running at the same time. This is ensured by the following integrity constraint; assumptions have been extended with starting time for most recent job and the guard refers to an auxiliary predicate that holds if and only of all three indicated time intervals have a point in common.

```
+printer(lexon2000,S1,F1), +printer(epsmark1993,S2,F2),
   +printer(pewhack2004,S3,F3) ==>
  overlapping((S1,F1),(S2,F2),(S3,F3))  |  fail.
```

(A few modifications of the scheduler predicate `run` is needed.)

Assumptions and expectation operators are implemented in CHR in a way similar to abduction, but need extra care for scoping and matching of expectations with assumptions. Each operator is implemented by one single-headed CHR rule that employs the constraint store as container in a straightforward procedural way. Consumed assumptions and expectations are removed when relevant by the `remove_constraint` device. In case of backtrack, they will be added back to the constraint store so that other matches can be tried out.

Assumptions made by "`*`" and "`+`" are just added to the store and an expectation by "`-`" will retrieve all possible assumptions in the store and set up

a control structure that can try out all of them upon backtracking. This is implemented by the following CHR rule and auxiliary predicate; in this case the expectation can be removed using a simplification rule as it does not need to be added on backtrack.

```
-B <=> findall_constraints(+C,PlusList),
       findall_constraints(*C,StarList),
       append(PlusList,StarList,List),
       choice(-B,List).

choice(-B,[(+A) # X | More]):-
        (remove_constraint(X), A=B ; choice(-B,More)).
choice(-B,[(*A) # _| More]):- (A=B ; choice(-B,More)).
```

As it appears, an intuitionistic assumption is removed when matched with an expectation (and potentially put back into the store upon backtracking); linear ones stay in the store so they can possibly apply to other expectations. Timeless assumptions and expectations are implemented in a similar, but slightly more complicated way since a new assumption must be compared with any possible, pending expectation. The straightforward code is given in the appendix.

4 Abduction and assumptions in DCGs

Language analysis is an area in which both abduction and assumptions have proved to be useful. Here we show how our versions of the two paradigms, separately and combined, work smoothly together with Definite Clause Grammars (DCGs) [16]; recall that most Prolog systems include this notion and compile it into Prolog when a source file is loaded.

Abduction for still life analysis

Abduction is especially useful for extracting information about the semantic context it which a discourse takes place. We show an example adapted from [5] that analyzes sentences about still life images such as *"The flower is on the table"* which can be interpreted in three different ways: that the flower is lying on the table or, what is the normal case, that the flower is placed in a container object (typically a vase) where the latter is in physical contact with the table (literally "on"), or finally that the flower is lying on an object which is placed on the table (may or may not be a container).

From the point of view of abductive interpretation, a sentence such as *"The flower is on the table"* can only by uttered truthfully if necessary facts (some or other) hold for the still life under consideration. Notice that the intuitive meaning of a nonterminal changes compared with a purely syntactic analysis: `sentence` refers to a true sentence and `thing` refers to a thing depicted in the given still life.

We use abducibles `!in/2` and `!on/s` to refer to immediate physical relationships, and `!thing` and `!container` to existence of given kinds of objects. The following set of integrity constraints describes general properties of the world.

```
!in(X,Y), !on(Y,X) ==> fail.    !in(X,X) ==> fail.
!on(X,Y), !in(Y,X) ==> fail.    !on(X,X) ==> fail.
!on(X,Y), !on(Y,X) ==> fail.    !container(C) ==> !thing(C).
!in(X,Y), !in(Y,X) ==> fail.    !in(the_box,the_vase) ==> fail.
!on(the_flower,_) ==> !not normal(the_flower).
!on(_,the_flower) ==> !very_flat(the_flower).
```

The grammar is as follows; notice that the arrow of the DCG notation goes in opposite direction of the logical implication.

```
sentence -->  thing(A), [is, on], thing(B),
  {!thing(A), !thing(B),
   (!on(A,B)
   ; ((!container(X),!in(A,X);!thing(X),!on(A,X)),!on(X,B)))}.

thing(T) --> [T], {!thing(T)}.
```

The analysis initiated by `phrase(sentence, [the_flower,is,on,the_table])` gives the three different interpretations indicated above.

Assumptions for coordination and anaphora

The following example has been adapted from [11, 7] and shows two applications of assumptions, for resolving pronoun references and for a simple coordination problem. In a sentence *"Peter likes her"* the pronoun is expected to stand for a female character who has been mentioned earlier in the discourse. The following rule defines how the mentioning of a proper name produces an assumption that makes the individual available for future reference.

```
np(X,Gender) --> name(X,Gender), {*acting(X,Gender)}
```

Assume for the moment the following rule for a sentence and sequences of sentences.

```
sentence(s(A,V,B)) -->  np(A,_), verb(V), np(B,_).
sentences((S1,S2)) --> sentence(S1),sentences(S2).
sentences(nil) --> [].
```

The following rules define how a pronoun can appear in a sentence with its meaning given by an expectation.

```
np(X,Gender) --> {-acting(X,Gender)}, pronoun(Gender).
pronoun(fem) --> [her].
```

The following query and answers show the behaviour.

```
?- phrase(sentences(S), [peter,likes,martha, mary,hates,her]).
S = (s(peter,like,martha),s(mary,hate,mary),nil) ? ;
S = (s(peter,like,martha),s(mary,hate,martha),nil) ? ;
no
```

The second answer expresses the interpretation we would expect, and the first one is an undesired consequence of the specification so far; we show it can be suppressed below.

The discourse *"Peter likes and Mary hates Martha"* contains two coordinating sentences in the sense that the first incomplete one takes its object from the second one. This can be described by having an incomplete sentence to put forward a timeless expectation that may be satisfied by a later assumption produced by a complete sentence; the following two grammar rules are sufficient.

```
sentence(s(A,V,B)) --> np(A,_), verb(V), np(B,_), {=*obj(B)}.
sentence(s(A,V,B)) --> np(A,_), verb(V), [and],   {=-obj(B)}.
```

Grammars with both abduction and assumptions

Abduction and assumptions can be mixed freely which we can use to provide a better solution to the pronoun resolution problem above. We modify the grammar rule for sentences such that semantic interpretation is made abductively, i.e., the sentence can be told honestly provided the semantic context contains the necessary facts.

```
sentence --> np(A,_), verb(V), np(B,_), {!s(A,V,B)}
!s(X,hate,X) ==> fail.
```

With this modification, the analysis of *"Peter likes Martha, Mary hates her"* gives only one solution. This grammar is interesting as it shows how different layers of analysis can assist each other, semantic knowledge about the hating relation is applied for guiding pronoun resolution.

It is also possible to have integrity constraints that refers to both abducibles and assumption/expectations at the same time although we have not developed interesting examples yet.

5 Conclusion and related work

We have described a methodology and implementation which make Abductive and Assumptive Logic Programming (A^2LP) available as useful programming paradigms with an execution speed comparable to traditional Prolog programs.

The approach avoids the heavy computational overhead associated with known metainterpreter based implementations of abduction (for instance, [13]) has the overhead of alternating abductive steps with resolution steps. The component of an A^2LP program that corresponds to a logic program is executed directly as a Prolog program, and its integrity constraints directly as CHR rules. This

means that A^2LP programs can be run through existing optimizing compilers for Prolog and CHR. It is interesting to point out that coroutining of integrity constraints is taken care of automatically by virtue of CHR rules.

There are existing, efficient implementations of Assumptive Logic Programs but the present work extends the paradigm with integrity constraints and the option to combine with abduction in a common framework.

The price paid for this efficiency and flexibility is a limitation on the use of negation. Some examples in the literature of abduction involving Event Calculus do not work in the approach but others, such as [19] on robot planning seems possible (has been implemented in CHR in an early experiment, but not tested in the present framework). The small examples in the present paper are intended to indicate that A^2LP covers a spectrum of interesting programs, and the fact that the paradigm can immediately be combined with any other constraint solver available in the Prolog version at hand substantiates this viewpoint.

The first observation of the similarity between CHR and abductive logic programming was made by [1] showing that abducible predicates can be represented as constraints in CHR's sense and integrity constraints as rules in CHR. The referenced work describes a translation of a class of abductive logic programs with limited use of negation (similar to the present paper) into $CHR^\vee$ [2] which is a languages the formalizes the use of disjunctions (Prolog semicolon) in rule bodies; the main difference is that [1] also translates the logic program component into $CHR^\vee$ so that the efficiency of having Prolog to do the resolution steps is lost. CHR based abduction for language processing is applied in the CHRG system (for CHR Grammars) [6, 7] which is based on bottom-up parsing in CHR.

A proposal for emulating abductive logic programming with assumptions was done in [10]. While less efficient than the present proposal, it allowed the same (abducible) predicate to be either proved normally, if this was possible, or abduced if not. It also put the ability to examine unconsumed assumptions to use in combining for instance defeasible reasoning with abductive logic programming, and in suggesting novel extensions such as conditional abductive logic programming—this latter, by abducing not only predicates, but also clauses.

The idea of abducing clauses was also useful in a grammatical context: [8] simulate abduction of grammar correction rules from a user defined input grammar in order to dynamically correct syntactic errors in sentences being analyzed; this approach is also described using CHR.

As we have noticed, negation is the more complicated part to which we have no solution; [3] sketches an extension of [1]'s method intended for a full use of negation-as-failure in program clauses and integrity constraints; as for [1], no integration with Prolog is provided. Unfortunately, it has not been possible to reconstruct the code from the description in [3] in order to test the method and there appears to inherent looping problems.

The Demo system described in [5] seems to be the first application of CHR to abduction and similar problems, in the shape of a general metainterpreter for logic programs which is reversible in the sense that it can generate programs that make specified goals provable; this property is made possible using a constraint

solver written in CHR for semantic primitives. In terms of efficiency this system is by no means comparable to what is described in the present paper.

Acknowledgements The authors want to thank Michael Cheng for experimentation with an early version of the methods and for helpful discussions. This work is supported by the CONTROL project, funded by Danish Natural Science Research Council; the first author is partly supported in part by the IT-University of Copenhagen, and the second author gratefully acknowledges support from Canada's NSERC Discovery Grant program.

Appendix: Source code for implementation of assumptions

```
:- use_module(library(chr)).     :- use_module(library(lists)).

:- op(500,fx,[*,=+,=-,=*]).

handler assumptions.
constraints =* /1, =- /1,=+ /1, * /1, - /1, + /1.

-B <=> findall_constraints(+C,PlusList),
       findall_constraints(*C,StarList),
       append(PlusList,StarList,List),
       choice(-B,List).

=-B  # ThisId ==>
        findall_constraints(=+_,PlusList),
        findall_constraints(=*_,StarList),
        append(PlusList,StarList,List),
        choice_timeless(=-B,ThisId,List).

=+B # ThisID ==>
   findall_constraints(=-C, List),
   choice_timeless(=+B, ThisID, List).

=*B # ThisID ==>
   findall_constraints(=-C, List),
   choice_timeless(=*B, ThisID, List).

expections_satisfied:- \+ find_constraint(-_,_), \+ find_constraint(=-_,_).

choice(-B,[(+A) # X | More]):- (remove_constraint(X), A=B ; choice(-B,More)).
choice(-B,[(*A) # _| More]):- (A=B ; choice(-B,More)).

choice_timeless(=-B,ThisId, [(=+A) # X | More]):-
   (remove_constraint(ThisId), remove_constraint(X), A=B
    ; choice_timeless(=-B,ThisId,More)).
```

```
choice_timeless(=-B,ThisId, [(=*A) # _ | More]):-
   (remove_constraint(ThisId), A=B
    ; choice_timeless(=-B,ThisId,More)).

choice_timeless(=+B,ThisId, [(=-A) # X | More]):-
   (remove_constraint(ThisId), remove_constraint(X), A=B
    ; choice_timeless(=+B,ThisId,More)).

choice_timeless(=*B,ThisId, [(=-A) # X | More]):-  % ThisId not used
   (remove_constraint(X), A=B ; true),
   choice_timeless(=+B,ThisId,More).

choice_timeless(_,_,[]):- true. % Puts back constraint and continues
```

References

1. Abdennadher, S., and Christiansen, H., An Experimental CLP Platform for Integrity Constraints and Abduction. Proceedings of FQAS2000, Flexible Query Answering Systems, pp. 141–152, *Advances in Soft Computing series,* Physica-Verlag (Springer), 2000.
2. Abdennadher, S., and Schütz, H. CHR$^{\vee}$: A flexible query language. Proc. FQAS'98, *Lecture Notes in Artificial Intelligence* 1495, pp. 1–14, Springer, 1998.
3. Badea, L., and Tilivea D.: Abductive Partial Order Planning with Dependent Fluents KI 2001: Advances in Artificial Intelligence, Joint German/Austrian Conference on AI, Baader, F., Brewka, G., Eiter, T., (eds.) *Lecture Notes in Artificial Intelligence* 2174 p. 63-77, Springer, 2001.
4. Chan, D., Constructive negation based on the database completion, *Proc. of Fifth International Conference and Symposium on Logic Programming*, (eds. Kowalski, Bowen), pp. 111–125, MIT Press, 1988.
5. Christiansen, H. Automated reasoning with a constraint-based metainterpreter, *Journal of Logic Programming*, Vol 37(1–3) Oct–Dec, pp. 213–253, 1998.
6. Christiansen, H., Abductive Language Interpretation as Bottom-up Deduction. In: Natural Language Understanding and Logic Programming, Proceedings of the 2002 workshop, ed. Wintner, S., *Datalogiske Skrifter* vol. 92, Roskilde University, Comp. Sci. Dept., pp. 33–47, 2002.
7. Christiansen, H., CHR grammars. *To appear in International Journal on Theory and Practice of Logic Programming, special issue on Constraint Handling Rules*, expected publication date medio 2005.
8. Christiansen, H., and Dahl, V., Logic Grammars for Diagnosis and Repair. *International Journal on Artificial Intelligence Tools*, Vol. 2, no. 3 (2003), pp. 227–248.
9. Dahl, V., and Tarau, P. From Assumptions to Meaning. *Canadian Artificial Intelligence* 42, Spring 1998.
10. . Dahl, V., and Tarau, P. *Assumptive Logic Programming.* Technical report, Simon Fraser University, 1999].
11. Dahl, V., Tarau, P., and Li, R., Assumption grammars for processing natural language. *Proc. Fourteenth International Conference on Logic Programming.* pp. 256–270, MIT Press, 1997.

12. Frühwirth, T.W., Theory and Practice of Constraint Handling Rules, *Journal of Logic Programming*, Vol. 37(1–3), pp. 95–138, 1998.
13. Kakas, A.C., Michael, A., and Mourlas, C. ACLP: Abductive Constraint Logic Programming, *The Journal of Logic Programming*, vol 44, pp. 129–177, 2000.
14. Kakas, A.A., Kowalski, R.A., and Toni, F. 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, pp. 235–324, 1998.
15. Muñoz-Hernández, S., Mariño, J., and Moreno-Navarro, J.J., Constructive Intensional Negation. In: Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Kameyama, Y., Stuckey, P.J., (eds.), *Lecture Notes in Computer Science* 2998; Springer-Verlag, pp. 39-54, 2004.
16. Pereira, F.C.N., and Warren, D.H.D., Definite clause grammars for language analysis. A survey of the formalism and a comparison with augmented transition grammars. *Artificial Intelligence* 10, no. 3–4, pp. 165–176, 1980.
17. Poole, D., Mackworth, A., and Goebel, R. *Computational Intelligence*, Oxford University Press, 1998.
18. *SICStus Prolog user's manual*. Version 3.11, SICS, Swedish Institute of Computer Science, 2004. Most recent version available at `http://www.sics.se/isl`.
19. Shanahan, M., Reinventing Shakey. *Logic-Based Artificial Intelligence*, Minker, J. (ed). Kluwer Academic, pp. 233–253, 1999.
20. Tarau, P., Dahl, V., and Fall, A. Backtrackable State with Linear Affine Implication and Assumption Grammars. In: Concurrency and parallelism, Programming, Networking, and Security, Jaffar, J. and Yap, R. (eds.). *Lecture Notes in Computer Science* 1179, Springer Verlag, pp. 53–64, 1996.