

# A short introduction to abduction in logic programming, its application to view update, and its implementation in CHR

Henning Christiansen

March 18, 2003

Database update via views can be considered and implemented as a sort of abductive reasoning. With Prolog and a little help from CHR (which is an extension of SICStus Prolog and others), we can illustrate how view updating can be performed — although this does not actually provide an implementation that interfaces with a database system.

This notion of abduction has nothing to do with kidnapping; the term was coined by philosopher C.S. Peirce (1839-1914) and we present here a simplified version of his ideas. Peirce's aim was to provide a theory about human reasoning, especially concerned with scientific discovery. He postulated three principles as *the* fundamental ones:

- **Deduction**, reasoning within the knowledge we have already, i.e., from those facts we know and those rules and regularities of the world that we are familiar with. E.g., reasoning from causes to effects:  
*“If you make a fire here, you are likely to burn down the house.”*
- **Induction**, finding general rules from the regularities that we have have experienced in the facts that we know; these rules can be used later for prediction:  
*“Every time I made a fire in my living room, the house burnt down, aha, ... the next time I make a fire in my living room, the house will burn down too”.*
- **Abduction**, reasoning from observed results to the basic facts from which they follow, quite often it means from an observed effect to produce a qualified guess for a possible cause:  
*“The house burnt down, perhaps that ##### had made a fire in the living room again.”*

We can replicate this in logic programming terms:

- A Prolog system is a purely deductive engine. It takes a program of rules and facts and it can calculate the logical consequences of that program.
- Induction is difficult; methods for so-called inductive logic programming has been developed, and by means of a lot of statistics and other complicated machinery, it synthesizes rules from collections of “facts” and “observations”.
- Abductive logic programming; roughly means from a claim of goal that is required to be true (i.e., being a consequence of the program), to extend to program with facts so that the goal becomes true.

Inductive logic programming has been successfully applied in molecular biology concerned with protein molecule shapes and human genealogy. Abduction has been applied to planning: A goal is to be achieved, e.g., getting a robot from one place to another, and the facts to abduced comprise the plan of small steps to be performed.

## A standard example of abduction in LP

Consider this Prolog program:

```
grass_is_wet:- rained_last_night.
grass_is_wet:- sprinkler_was_on.
```

It has two rules and no facts. Obviously the following query fails:

```
?- grass_is_wet.
no
```

An abductive interpreter does not give up that easily. It will do what it can to enforce that the query succeeds, and the only way it can do so is by suggesting which facts to add to the program.

A typical abductive interpreter do not actually modify the program source text but produces an *abductive answer* consisting of those facts, that if they were added to the program would make the query succeed. With an abductive interpreter, we may have a dialogue with the program above as follows:

```
?- grass_is_wet.
rained_last_night ? ;
sprinkler_was_on ? ;
no
```

For this play to make sense, it is necessary to specify which predicates are *defined* and which are *abducible*. A defined predicate is one defined by program clauses (similar to a database view) and abducible ones are those defined by facts only and that the interpreter is allowed to extend. We may define abduction so that the interpreter also can suggest deletion of facts to make the conclusion hold; we leave that out in this presentation.

Basically, an abductive interpreter works in the way that when it enters an abducible goal that otherwise would fail in Prolog, it simply notes that goal as part of the abductive answer.

In general, we also need *integrity constraints* in order to rule out “weird” abductive answers, and these integrity constraints can be understood intuitively and in their precise semantics as in the database context.

## Constraint handling rules

... or CHR is a recent extension to Prolog that allows the programmer to write his or her own constraint solvers. This notion of “constraints” has nothing to do with “integrity constraints” and there is an obvious confusion. Constraints in this sense refers to condition on variables such as “ $x > 7$ ”, and there is a long tradition for so-called constraint programming which has been applied in operation analysis, economics, etc.

Constraint *logic* programming is an approach to integrate the solving of such constraints with the normal execution of a logic program. There exist extensions of Prolog with such abilities, e.g., to provide a more satisfactory treatment of arithmetic than what is provided by Prolog’s “is”. We will not go into details or give examples here; we mention this simply to motivate origins and terminology of CHR.

As described, a constraint solver is a sort of black box (programmed in some non-obvious way as part of the underlying machinery), but CHR provides a language for programming about these things in a Prolog-like language. So to implement a constraint solver for real number arithmetic, the programmer can implement a general equation-and-inequation solver as a CHR program, typically using some sort of Gauß elimination procedure.

In a CHR program, certain predicates are declared as **constraints** which means that they are treated in a special way by the underlying system. CHR extends Prolog’s execution state with a *constraint store*. When a constraint is called as a goal (or subgoal in a clause body), it is added to the constraint store and if possible, constraint handling rules applies and transform constraint store.

We consider such rules in a moment; here we concentrate how the constraint store serves as a perfect container for abducibles. Consider the following program that combines Prolog and CHR; the first line declares two predicates as constraints so that they will be treated as indicated:

```

handler garden_humidity1.

constraints rained_last_night/0, sprinkler_was_on/1.

grass_is_wet:- rained_last_night.
grass_is_wet:- sprinkler_was_on.

```

When this program is executed, the constraints entered are added to the constraint store; handling of constraints is integrated with Prolog's backtracking as to avoid mixing up different constraint stores. The declaration `handler garden_humidity.` is not important, but CHR's syntax requires such a declaration.

The resulting constraint store is printed as part of the answer and we get exactly the following when asking a query for `grass_is_wet`.

```

?- grass_is_wet.
rained_last_night ? ;
sprinkler_was_on ? ;
no

```

This straightforward way of using CHR provides a first implementation of abduction. This was not intended the inventors of CHR, but the principle was identified by Abdennadher & Christiansen, FQAS 2000 ;-)

## Integrity constraints in CHR

CHR contains a repertoire of different rules for transforming the constraint store, but we need here only a small subset consisting of so-called propagation rules. Let us extend the program above with one more abducible predicate and an simple constraint handling rule that serves as an integrity constraint.

```

handler garden_humidity2.

constraints rained_last_night/0, sprinkler_was_on/1,
           full_moon_last_night/0.

rained_last_night, full_moon_last_night ==> fail.

grass_is_wet:- rained_last_night.
grass_is_wet:- sprinkler_was_on.

```

The propagation rule reads: If the constraint store contains the two constraints indicated in its left side, then its body (following the arrow) is

executed and in this case giving an explicit failure. This means that a constraint store  $\approx$  an abductive answer containing both `rained_last_night` and `full_moon_last_night` is not acceptable (and removed by backtracking).

```
?- full_moon_last_night, grass_is_wet.  
full_moon_last_night,  
sprinkler_was_on ? ;  
no
```

Integrity constraints can contain any executable expression as its body, in particular another abducible. Consider the following

```
handler garden_humidity2.  
  
constraints rained_last_night/0, sprinkler_was_on/1,  
            full_moon_last_night/0, mixed_weather_last_night/0.  
  
rained_last_night, full_moon_last_night ==> mixed_weather_last_night.  
  
grass_is_wet:- rained_last_night.  
grass_is_wet:- sprinkler_was_on.
```

This results in the following:

```
?- full_moon_last_night, grass_is_wet.  
rained_last_night, mixed_weather_last_night ? ;  
sprinkler_was_on ? ;  
no
```

## View update

The following program combining Prolog and CHR defines a little database with integrity constraints.

```
:- use_module(library(chr)).  
handler view_update.  
  
constraints father/2, mother/2.  
  
father(X,Y), father(Z,Y) ==> Z=X.  
mother(X,Y), mother(Z,Y) ==> Z=X.  
  
grandfather(X,Z):- father(X,Y), father(Y,Z).  
grandfather(X,Z):- father(X,Y), mother(Y,Z).
```

```
current_db:-
    father(peter, paul),
    father(paul, jens),
    mother(marie, jens).
```

It may seem a bit confusing that the current database is not defined by means of facts as usual, but the strange way above is needed in order to provide an interaction between new and old database facts.

If we had listed the current database as Prolog facts, it would be not possible for the integrity constraints to see them (and the semantics of the whole program would be unclear).

In order to perform a view update to a given database, we need mention it in the database; notice that we get the the whole updated database as answer.

```
?- current_db, grandfather(X, jens).
X = peter,
father(peter, paul),
father(paul, jens),
mother(marie, jens),
father(peter, paul),
father(paul, jens) ? ;

father(peter, paul),
father(paul, jens),
mother(marie, jens),
father(X, marie),
mother(marie, jens) ? ;
no
```

As it appears, the last answer contains a variable **X** in the position of a new **father** for **marie**; this means that any value for **X** is acceptable — provided that it does not introduce an inconsistency (but with no detailed explanation of what this means).

Finally, we will mention that this Prolog+CHR prototype of view updating only is realistic in the way it uses rules backwards in order to translate a view update into updates of tabular relations, but integrity checking should, of course, be done in another way, involving simplified integrity constraints.

## Difficulties

The following phenomena in integrity constraints are difficult to handle when implementing abduction in CHR as indicated.

- Intensional predicates; if they are nonrecursive, they can be unfolded to a combination on extensional (i.e., abducibles) but in general it is not clear how to treat them.
- Negation; negated abducibles can be handled by so-called explicit negation (not explained here), but negated, intensional predicates are difficult.

## References

Search or ask your teacher :)