Course on Artificial Intelligence and Intelligent Systems

A short introduction to CHR and its application for rule-based expert systems

A course note

Henning Christiansen
Roskilde University, Computer Science Dept.
 Version 12 sep 2005

Contents

1 Introduction and overview

The language of Constraint Handling Rules, CHR, is an extension to Prolog intended as a declarative language for writing constraint solvers for CLP systems; here we give a very compact introduction and refer to [5] for a more formal and covering presentation. Examples have been developed using the SICStus Prolog version of CHR and we refer the reader to its manual [7] for remaining technicalities. As textbook on CHR and constraint programming, we refer to [2]. CHR is now integrated in several major Prolog implementations and has gained popularity for a variety of applications due to its expressibility and flexibility, which goes far beyond the traditional applications of constraint programming (such as finite domains, arithmetic, etc.). We refer to the CHR web site [1] for further information on CHR, including its different implementations and applications.

Prolog has its fixed top-down or backward chaining strategy for program execution and CHR extends with bottom-up or forward chaining, and in this way, the combination of CHR and Prolog provides a powerful environment for illustrating the basic mechanisms of rule-based expert systems. In fact, CHR itself allows for quite flexible combinations of the two execution strategies, but we consider here also the combination with Prolog, the latter kept responsible for backward chaining.

Overview

Section 2 introduces the fundamentals of CHR by means of examples and without too many formal details; references for those who want the full story were given above. In section 3, we introduce a little knowledge base and show how it can be written as a Prolog program which makes it behave as a backward-chaining expert system. Section 4 shows how forward chaining can be obtained by writing the knowledge base as CHR propagation rules, and in section 5 [[NOT INCLUDED YET]] we show how the two modes can be combined.

A knowledge of Prolog programming up to the level of [3] is assumed. All examples are available online as program files that run under SICStus Prolog; see course web pages [4].

2 Basic CHR

CHR takes over the basic syntactic and semantic notions from Prolog and extends with its specific kinds of rules. The execution of CHR programs is based on a *constraint store*, and the effect of applying a rule is to change the effect of the store. For a program written in the combination of Prolog and CHR, the system switches between two tow. When a Prolog goal is called, it is executed in the usual top-down (or goal-directed) way, and when a Prolog rule calls a CHR constraint, this will be added to the constraint store — then the CHR rules apply as far as possible, and control then returns to the next Prolog goal.

Technically speaking, constraints of CHR are first-order atoms whose predicates are designated constraint predicates, and a constraint store is a set of such constraints, possible including variables that are understood existentially quantified at the outermost level. A constraint solver is defined in terms of rules which can be of the following two kinds.

$$\text{Simplification rules: } c_1, \ldots, c_n \texttt{ <=> } \mathit{Guard} \mid c_{n+1}, \ldots, c_m$$
$$\text{Propagation rules: } c_1, \ldots, c_n \texttt{ ==> } \mathit{Guard} \mid c_{n+1}, \ldots, c_m$$

The c's are atoms that represent constraints, possible with variables, and a simplification rule works by replacing in the constraint store, a possible set of constraints that matches the pattern given by the *head* $c_1, \ldots c_n$ by those corresponding constraints given by the *body* $c_{n+1}, \ldots, c_m$, however only if the condition given by *Guard* holds. A propagation rule executes in a similar way but without removing the head constraints from the store. What is to the left of the arrow symbols is called the *head*[1] and what is to the right of the guard the *body*. The declarative semantics is hinted by the applied arrow symbols (bi-implication, resp., implication formulas, with variables assumed to be universally quantified) and it can be shown that the indicated procedural semantics agrees with this. This is CHR explained in a nutshell.

CHR provides an third kind of rules, called *simpagation rules*, which can be thought of as a combination of the two or, alternatively, as an abbrevation for a specific form of simplification rules.

$$\text{Simpagation rules: } c_1, \ldots, c_i \setminus c_{i+1}, \ldots c_n \texttt{ <=> } \mathit{Guard} \mid c_{n+1}, \ldots, c_m$$
$$\text{which can be though of as: } c_1, \ldots, c_n \texttt{ <=> } \mathit{Guard} \mid c_1, \ldots, c_i, c_{n+1}, \ldots, c_m$$

In other words, when applied, $c_1, \ldots, c_i$ stays in the constraint store and $c_{i+1}, \ldots, c_n$ are removed.

In practice, the body of CHR rules can include any executable Prolog expression including various control structures and calls to Prolog predicates. Similarly, Prolog rules and queries can make calls to constraints which, then, may activate the CHR rules.

The guards can be any combination of predicates (built-in or defined by the programmer) that test the variables in the head, but in general guards should not change values of these variables or call other constraints; it these cases, the semantics gets complicated, see references given above if you may have interest in the details. Finally, guards can be left out together with the vertical bar, corresponding to a guard that always evaluates to true.

The following example of a CHR program is adapted from the reference manual [7]; from a knowledge representation point of view it may seem a bit strange, but it shows the main ideas. It defines a a little constraint solver for a single constraint `leq` with the intuitive meaning of less-than-or-equal. The predicate is declared as an infix operator to enhance reading, but this is not necessary (`X leq Y` could be written equivalently as `leq(X,Y)`).

```
:- use_module(library(chr)).
handler leq_handler.
constraints leq/2.
:- op(500, xfx, leq).
X leq Y , Y leq Z ==> X leq Z.
X leq Y , Y leq X <=> X=Y.
X leq X <=> true.
X leq Y \ X leq Y <=> true.
```

The first line loads the CHR library which makes the syntax and facilities used in the file available. The `handler` directive is not very interesting but is required. Next, the constraint predicates are declared as such (here only one such predicates) and this informs the Prolog system that occurrences of these predicates should be treated in a special way.

[1] Some authors call each constraint to the left of the arrow a head, and with that terminology, CHR has multi-headed rules.

The program consists of four rules, one propagation, two simplifications, and one simpagation. The first simplification describe transitivity of the `leq` constraints. If, for example, the constraints `a leq b` and `b leq c` are called, this rule can fire and will produce a new constraint `a leq c` (which in turn may activate other rules).

The second rule as a simplification rule which will remove the two constraints and unify the arguments. Intuitively, the rule says that if some `X` is less than or equal to some `Y` and the reverse also holds, then they should be considered equal. With constraint store {`a leq Z`, `Z leq a`}, the rule can apply, removing the two constraints and unifying variable `Z` with the constant symbols `a`.

Consider a slightly different example, the constraint store {`a leq b`, `b leq a`}. Again, the rule can apply, removing the two constraints from the store and calling `a=b`. This will fail as `a` and `b` are two different constant symbols.

Notice that CHR is a so-called *committed choice* language in the sense that when a rule has been called, a failure as exemplified above will not result in backtracking. I.e., in the example, the observed failure will **not** add {`a leq b`, `b leq a`} back to the constraint store so other and perhaps more successful rules may be tried out. However, when CHR is combined with Prolog, a failure as shown will cause Prolog to backtrack, i.e., it will undo the addition of the last of the two, say `b leq a`, and go back to the most recent choice point.

The simplification rule `X leq X <=> true` will remove any `leq` constraint from the store with two identical arguments. This illustrates a fundamental difference between Prolog and CHR. Where Prolog uses unification when one of its rules is applied to some goal, CHR uses co-called matching. This means that the mentioned rule will apply to `a leq a` but not to `a leq Z`. In contrast, the application of Prolog rule `p(X,X):-···` to `p(a,Z)` will result in `a=Z` before the body is entered.

The final rule in the program above is a simpagation rule `X leq Y \ X leq Y <=> true` which serves the purpose of removing duplicate constraints from the store.

We will consider the following query and see how the constraint store changes.

```
?- C leq A, B leq C, A leq B.
```

Calling the first constraint triggers no rule and we get the constraint store {`C leq A`}. Calling the next one will trigger the transitivity rule (the first rule), and we get {`C leq A`, `B leq C`, `B leq A`}. The last call in the query will trigger a sequence of events. When `A leq B` is added to the constraint store, it reacts, so to speak, with `B leq A` and the second rule applies, removing the two but resulting in the unification of `A` and `B`; let us for clarity call the common variable `V1` which is referred to by both `A` and `B`. Now the constraint store is {`C leq V1`, `V1 leq C`}. Now the same rule can apply once again, unifying `C` and `V1`, so that the result returned for the query is the empty constraint store and the bindings `A=B=C`.

In general, when a query is given to a CHR program (or a program written in the combined language of CHR plus Prolog), the system will print out the final constraint store together with Prolog's normal answer substitution. Alternative solution can be asked for as in traditional Prolog by typing semicolon.

3 Warming up: A little knowledge base and expert system in Prolog

3.1 The example knowledge base

Consider this little knowledge base written as a set of if-then rules (see [6] for an introduction to such rules).

1. If rains and go_out and have_not(umbrella) then get_wet.
2. If shower_bath then get_wet.
3. If need(X) and have_not(X) then go_out and buy(X).
4. If thirsty then need(beer).

We can illustrate backwards reasoning by an investigation of whether a user will get wet. Rule 2 provides one way of getting wet, so the system may go from conclusion to premise and ask the user "Will you take a shower?" and if user replies "yes", the system can reply "Well, then you'll get wet." Trying instead using rule 1 in backwards direction, may result in questions whether it rains, and via backwards application of rule 3 followed by 4 whether the user is thirsty, and finally if he has no umbrella; in that case, the user will also get wet when he goes out to buy beer.

These rules can be rewritten into Prolog by considering each piece of data as a predicate, for example the second rule as `get_wet:- shower_bath.` This would not be very useful, however, as primitive information such as "thirsty" (etc.) needs to be added as a fact `thirsty.` to the program. A dialogue with the user seems difficult.

Below we will introduce a little expert system shell implemented in a few lines of Prolog that makes it easy to use Prolog for backward chaining expert systems which provides a dialogue with the user

3.2 A simple expert system shell in Prolog

The file `expert0` contains Prolog definitions which makes it possible to read in knowledge bases in a format illustrated by sample file below. Rules are written as Prolog rules in the standard way with conclusion to the left and premises to the right.

The system inherits Prolog's goal-directed (top-down, backward chaining) strategy and uses a "generic" predicate called `goal` as a container for all data. Notice that rules, such as rule 3 above, with "and" in the conclusion needs to be rewritten into more that one rule.

To use this system you should load it into Prolog as follows:

```
:- [expert0].
```

Now you can load in your own source files; e.g.,

```
:- [my_kb0].
```

The following shows the full source text for an expert system which could be the file `my_kb0`.

```
:- multifile goal/1.

goal(get_wet):- goal(rains), goal(go_out), goal(have_not(umbrella)).
goal(get_wet):- goal(shower_bath).
goal(go_out):- goal(need(X)), goal(have_not(X)).
goal(buy(X)):- goal(need(X)), goal(have_not(X)).
goal(need(beer)):- goal(thirsty).

can_question(rains).
can_question(thirsty).
can_question(have_not(_)).
can_question(shower_bath).
```

The first line is a little Prolog technicality which is needed since the file `expert0` contains a Prolog clause for the `goal` predicates that provides the dialogue with the user.[2]

The next lines constitute the rules of the knowledge base, and the last bunch of lines defines which data the system can query the user about.

It is possible to query a knowledge base by a standard Prolog query using the `goal` predicate but for some subtle reason you cannot investigate different possible ways that made the given goal succeed. Use instead the predicate `test_goal`; the following exemplifies a dialogue with the system.

```
| ?- test_goal(get_wet).
Is it true that rains?y
Is it true that thirsty?y
Is it true that have_not(beer)?y
Is it true that have_not(umbrella)?y
get_wet is true. Do you want me to check other ways?y
Is it true that shower_bath?y
get_wet is true. Do you want me to check other ways?y
no
| ?- test_goal(get_wet).
Is it true that rains?n
Is it true that shower_bath?y
get_wet is true. Do you want me to check other ways?n
yes
```

3.3 Adding explanations

In may be complained that the little backward chaining expert system above lacks one important feature in order to be called an expert system, namely that of producing an explanation for how it proved the given goal.

However, this can easily be included by adding an extra argument to the goal predicate, which holds an explanation for that specific goal. When the user is asked for a goal and tell it to succeed, we add the explanation `user`, so for example when asked if he is thirsty, the goal succeeds as `goal(thirsty,user)`.

[2]Normally, and without the `multifile` directive, all clauses for a given predicate must be in one and the same file.

Each rule in the knowledge base is then extended so that it constructs an explanation for its conclusion from its subgoals and their explanations. For example:

```
goal(go_out,E):- goal(need(X),E1), goal(have_not(X),E2),
    E = [[need(X),E1], [have_not(X),E2]]
```

The explanation returned for, say, `goal(get_wet, Exp)` will then be a huge term with several levels of lists-in-lists that gives the full explanation. We will not study generation of explanations in more detail, and our purpose here has been simply to demonstrate that it straightforward to generate them.

4 Forward chaining expert system in CHR

4.1 Propagation rules are forward chaining

Propagation rules in CHR provide a natural paradigm for forward chaining. From given sets of input contraints representing given data, the rules will apply as long as possible, deriving all possible consequences thereof.

As with Prolog above, we can in principle defines each possible fact as a constraint, but for simplicity and the developments to follow, we prefer instead to introduce a "generic" constraint to hold facts. A forward chaining version of our sample knowledge base can be written directly as a CHR program that needs no extra definitions, i.e., the file can be run alone.

```
:- use_module(library(chr)).
handler forward_chaining.
constraints fact/1.

fact(rains), fact(go_out), fact(have_not(umbrella)) ==> fact(get_wet).
fact(shower_bath) ==> fact(get_wet).
fact(need(X)), fact(have_not(X)) ==> fact(go_out), fact(buy(X)).
fact(thirsty) ==> fact(need(beer)).
```

Notice that CHR's syntax allows us to have multiple facts as both premise and conclusion. To investigate the consequences of a given situation, we simply enter all known primitive facts as a query, and the system calculates the set of all consequences. Here is an example.

```
| ?- fact(thirsty), fact(rains), fact(have_not(beer)), fact(have_not(umbrella)).
fact(thirsty),
fact(need(beer)),
fact(rains),
fact(have_not(beer)),
fact(go_out),
fact(buy(beer)),
fact(have_not(umbrella)),
fact(get_wet) ?
yes
```

We observe that `get_wet` is reported in the final state oprinted out, so the system have told us that in the situation described, being thirsty, no beer in the house and no umbrella, on a rainy day, we well eventually get wet.

It is, of course, possible to extend the program above so its presents a more user friendly dialogue and only prints out those new facts that are derived, but we leave this out in order not to destroy the clarity of the presentation. Explanations can easily be added similarly to what we described in section 3.3, and we shall not consider this topic further.

4.2 Conflict resolution

As described in Negnevitsky's book [6], section 2.8, there may be conflicts among the rules of a knowledge base, so that in some situations, two different rules can apply, leading to different conclusions that are considered inconsistent with each other. In such cases, the expert system should have some strategy to decide which of the two rules to apply. Such principles do not fit very well into CHR, as the meaning of a set of CHR rules is defined in terms of first order predicate logic in which there is no sense in consider some rules better that others.

Let us consider a little example for crossing the street, and assume the following four CHR rules.

```
fact(light(green))        ==> fact(action(go)).
fact(light(red))          ==> fact(action(wait)).
fact(car_in_full_speed)   ==> fact(action(wait)).
fact(no_traffic)          ==> fact(action(go)).
```

CHR allows us to specify explicitly what are the inconsistencies, which here could be that the colour of the light is unique and the same for the action.

```
fact(ligth(X)),  fact(ligth(Y))  ==> X=Y.
fact(action(X)), fact(action(Y)) ==> X=Y.
```

Such rules corresponds to what in databases and abductive logic programming are called *integrity constraints*. Consider then the query from which a conflict arises.

```
?- fact(light(red)), fact(no_traffic).
```

These two fact will lead to the derived facts `fact(action(wait))` and `fact(action(go))`; next, these two trigger the second integrity constraint, resulting in an attempt of the unification `wait=go` which obviously fails. Thus the answer to the query is a useless "`no`".

So the conclusion must be that CHR is well suited for representation of consistent knowledge bases and for reasoning about consistent states, but additional complications such as avoidance of inconsistencies by priorities does not fit it.

Of course, CHR is a general programming language and integrated with Prolog so it will be possible to whatever conflict resolution a designer has in mind, but we may loose the transparent reading of the rules. We will, however, indicate some possible strategies which could be developed further in student projects.

4.2.1 Preference to some facts over others

In the example above we could make the integrity constraints less hard. It might be obvious here, in case of conflicts among possible actions to prefer the one which seems the most safe, i.e., prefer `action(wait)` for `action(go)`, and if there were conflicting information about the colour of the light, keep the conflict but issue a warning.

```
fact(ligth(X)),  fact(ligth(Y))  ==>
                          X \== Y | write('Warning: Colour of light not unique').
fact(action(wait)) \ fact(action(go)) ==> true.
```

You should be very careful when using CHR in this way. Assume that `fact(action(go))` happen to be the first fact to be derived, and that it via other rules, leads to the creation of other facts, call them F. If now, later, `fact(action(wait))` is derived, then the modified integrity constraint above will remove `fact(action(go))`, and thus the total program should be written in such a way that the facts of F also are removed.

4.2.2 Ad-hoc weights in CHR knowledge bases

We may consider adding ad-hoc numerical weights to each rule, that are used for determining a weight for each fact. The weight of a fact derived via a rule may then be the product of the two numbers, from the fact to which the rule is applied, and the weight assigned to the given rule. It will be practical to have all numbers to be between 0 and 1. The rule base above may be extended with weights as follows.

```
fact(light(green),V)      ==> W is V * 0.9, fact(action(go),W).
fact(light(red),V)        ==> W is V * 0.9, fact(action(wait),W).
fact(car_in_full_speed,V) ==> W is V * 1,   fact(action(wait),W).
fact(no_traffic,V))       ==> W is V * 1,   fact(action(go),W).
```

We see that the designer has given highest priority to those rule that depends on the actual traffic rather than the traffic lights. Integrity constraints can then be modified so that they resolve conflicts be throwing away the one out of two conflicting facts which has the lowest priority. This can be done by simpagation rules as follows.

```
fact(ligth(X),V)  \ fact(ligth(Y),W)  <=> V >= W | true.
fact(action(X),V) \ fact(action(Y),W) <=> V >= W | true.
```

In this way the query

```
?- fact(light(red),1), fact(no_traffic,1).
```

would lead to the answer that you should walk (with weigh 1).

We have indicated here how priorities may be added to CHR knowledge bases, but we will stress that this was done in a completely ad-hoc manner. If you want to add priorities to knowledge bases written in CHR, it is suggested that base you approach on a firm theoretical basis such as probability theory or fuzzy logic. The remarks made for the previous approach should be repeated also here, that when you explicitly remove an undesired fact, you should take care also to remove those other facts derived from it.

5 Combining forward and backward chaining

The text book by Negnevitsky [6] indicates, p. 40 at the bottom, that forward and backward chaining can be combined. Backward or goal directed is the most convenient when asking queries to the currently known set of facts, and forward chaining is used when a new fact is recognized so that the fact base is as complete as possible.

It is possible with a few additional "linking rules" to combine the two versions of a knowledge base that we have shown, backward chaining using Prolog and forward chaining using CHR, to achieve this effect. It does not seem very instructive to show the details here so we shall avoid this, as it amounts only to a small optimization of the backward chaining version (but no essential new functionality).

However, if we consider adding integrity constraits, conflict resolution and/or priorities to the forward chaining part, the combination may become interesting.

6 Exercises

Exercise 1 Identify some problem field of which you have good knowledge, and write down some if-then rules about that domain similarly to those of section 3.1.

Implement the rules as a forward chaining CHR program as indicated in section 4.

Exercise 2 Extend the knowledge base that you wrote down in CHR for the previous exercise with integrity constraints and perhaps additional rules as to make conflicts possible. Apply some of the principles sketched in section 4.2 in order to resolve conflicts. (You may choose to avoid some of the technical caveats given, and simply test out different approaches and see how they work).

References

[1] The programming language CHR, Constraint Handling Rules; official web pages. `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`

[2] Slim Abdennadher and Thom Frühwirth. *Essentials of Constraint Programming*. Springer, 2003.

[3] Henning Christiansen. Introduction to Prolog as a database language. A course note, 2003. `http://www.ruc.dk/~henning/KIIS05/DatabaseProlog.pdf`

[4] Henning Christiansen. Artificial intelligence and intelligent systems (KIIS). Course web pages, 2005. `http://www.ruc.dk/~henning/KIIS05/`

[5] 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.

[6] Michael Negnevitsky. *Artificial Intelligence, A Guide to Intelligent systems*. Addison-Wesley, 2nd edition, 2004.

[7] Swedish Institute of Computer Science. SICStus Prolog user's manual, Version 3.12. Most recent version available at `http://www.sics.se/isl`, 2004.