GWLP 94. Ref
Sgl 21 94
Tutorial Früh

Constraint Handling Rules*

Extended Abstract

Thom Frühwirth
ECRC, Arabellastrasse 17, D-81925 Munich, Germany
email: thom@ecrc.de

Abstract

Constraint handling rules (CHRs) are a high level language (extension) for writing constraint systems. The full version of this paper appears in [Fru95].

1 Introduction

Constraint logic programming [JaLa87, VH89, HS90, VH91, F*92, JaMa94] combines the advantages of logic programming and constraint solving. In logic programming, problems are stated in a declarative way using rules to define relations (predicates). Problems are solved by the built-in logic programming engine (LPE) using chronological backtrack search. In constraint solving, efficient special-purpose algorithms are employed to solve problems involving distinguished relations referred to as constraints.

A practical problem remains: Constraint solving is usually 'hard-wired' in a built-in constraint solver (CS) written in a low-level language. While efficient, this approach makes it hard to modify a CS or build a CS over a new domain, let alone verify its correctness. We proposed *constraint handling rules* (CHRs) [Fru92, Fru93a, Fru93b, Fru94, B*94, Fru95, FrHa95] to overcome this problem. CHRs are a language *extension* providing a declarative means to introduce *user-defined* constraints into a given high-level host language. In this extended abstract the host language is Prolog, a CLP language with equality over Herbrand terms as the only built-in constraint.

CHRs define *simplification* of and *propagation* over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence, e.g.

```
X>Y,Y>X <=> false.
```

Propagation adds new constraints which are logically redundant but may cause further simplification, e.g.

```
X>Y,Y>Z ==> X>Z.
```

*Part of this work is supported by ESPRIT Project 5291 CHIC.

When repeatedly applied by a constraint handling engine (CHE) the constraints are incrementally solved as in a CS, e.g.

`A>B,B>C,C>A` results in `false`.

CHIP was the first CLP language to introduce constructs (demons, forward rules, conditionals) [VH89] for user-defined *constraint handling* (like constraint solving, simplification, propagation). These various constructs have been generalized into CHRs. CHRs are based on guarded rules, as can be found in concurrent logic programming languages [Sha89], in the Swedish branch of the Andorra family [HaJa90], Saraswats cc-framework of concurrent constraint programming [Sar93], and in the 'Guarded Rules' of [Smo91]. However all these languages (except CHIP) lack features essential to define non-trivial constraint handling, namely for handling conjunctions of constraints and defining constraint propagation. CHRs provide these two features using multi-headed rules and propagation rules.

2 Constraint Handling Rules

Here we assume that constraint handling rules extend a given constraint logic programming language. The syntax and semantics given here reflect this choice. It should be stressed, however, that the host language for CHRs need not be a CLP language. Indeed work has been done at DFKI in the context of LISP [Her93]. This section follows [FrHa95].

2.1 Syntax

A CLP+CH program is a finite set of clauses from the CLP language and from the language of CHRs. Clauses are built from atoms of the form $p(t_1, ...t_n)$ where p is a predicate symbol of arity n ($n \geq 0$) and $t_1, ...t_n$ is a n-tuple of terms. A term is a variable, e.g. X, or of the form $f(t_1, ...t_n)$ where f is a function symbol of arity n ($n \geq 0$) applied to a n-tuple of terms. Function symbols of arity 0 are also called constants. Predicate and function symbols start with lowercase letters while variables start with uppercase letters. Infix notation may be used for specific predicate symbols (e.g. $X = Y$) and functions symbols (e.g. $-X + Y$). There are two classes of distinguished atoms, built-in constraints and user-defined constraints. In most CLP languages there is a built-in constraint for syntactic equality over Herbrand terms, `=`, performing unification. The built-in constraint `true`, which is always satisfied, can be seen as an abbreviation for `1=1`. `false` (short for `1=2`) is the built-in constraint representing inconsistency.

A *CLP clause* is of the form

$$H :- B_1, \ldots B_n. \quad (n \geq 0)$$

where the head H is an atom but not a built-in constraint, the body $B_1, \ldots B_n$ is a conjunction of literals called *goals*. The empty body ($n = 0$) of a CLP clause may be denoted by the built-in constraint `true`. A *query* is a CLP clause without head.

There are two kinds of CHRs[1]. A *simplification* CHR is of the form

$$H_1, \ldots H_i \texttt{ <=> } G_1, \ldots G_j \mid B_1, \ldots B_k.$$

A *propagation* CHR is of the form

$$H_1, \ldots H_i \texttt{ ==> } G_1, \ldots G_j \mid B_1, \ldots B_k.$$

A *label declaration* for a user-defined constraint H is of the form

label_with H if $G_1, \ldots G_j$

where $(i > 0, j \geq 0, k \geq 0)$ and the multi-head $H_1, \ldots H_i$ is a conjunction of user-defined constraints and the guard $G_1, \ldots G_j$ is a conjunction of literals which neither are, nor depend on, user-defined constraints.

2.2 Semantics

Declaratively, CLP programs are interpreted as formulas in first order logic. Extending a CLP language with CHRs preserves its declarative semantics. A CLP+CH program P is a conjunction of universally quantified clauses.

A CLP clause is an implication[2]

$$H \leftarrow B_1 \wedge \ldots B_n.$$

A simplification CHR is a logical equivalence provided the guard is true in the current context

$$(G_1 \wedge \ldots G_j) \rightarrow (H_1 \wedge \ldots H_i \leftrightarrow B_1 \wedge \ldots B_k).$$

A propagation CHR is an implication provided the guard is true

$$(G_1 \wedge \ldots G_j) \rightarrow (H_1 \wedge \ldots H_i \rightarrow B_1 \wedge \ldots B_k).$$

Procedurally, a CHR can fire if its guard allows it. A firing simplification CHR *replaces* the head constraint by the body, a firing propagation CHR *adds* the body to the head constraints. No theorem proving in the general sense is required to reason with the formulas expressed by CHRs.

The *operational semantics* of CLP+CH can be described by a transition system.

A *computation state* is a tuple

$$< Gs, C_U, C_B >,$$

where Gs is a set of goals, C_U and C_B are constraint stores for user-defined and built-in constraints respectively. A *constraint store* is a set of constraints. A set of atoms represents a conjunction of atoms.

The *initial state* consists of a query Gs and empty constraint stores,

$$< Gs, \{\}, \{\} >.$$

[1] A third, hybrid kind as well as options and declarations are described in [B*94].

[2] For simplicity of presentation, we do not use Clark's completion here.

A *final state* is either *failed* (due to an inconsistent built-in constraint store represented by the unsatisfiable constraint `false`),

$$< Gs, C_U, \{\texttt{false}\} >,$$

or *successful* (no goals left to solve),

$$< \{\}, C_U, C_B >.$$

The union of the constraint stores in a successful final state is called *conditional answer* for the query Gs, written $answer(Gs)$.

The built-in constraint solver (CS) works on built-in constraints in C_B and Gs, the user-defined CS on user-defined constraints in C_U and Gs using CHRs, and the logic programming engine (LPE) on goals in Gs and C_U using CLP clauses. The following *computation steps* are possible to get from one computation state to the next.

Solve
$< \{C\} \cup Gs, C_U, C_B > \longmapsto < Gs, C_U, C'_B >$
if $(C \wedge C_B) \leftrightarrow C'_B$

The built-in CS updates the constraint store C_B if a new constraint C was found in Gs. To *update* the constraint store means to produce a new constraint store C'_B that is logically equivalent to the conjunction of the new constraint and the old constraint store.

We will write $H =_{set} H'$ to denote equality between the sets H and H', i.e. $H = \{A_1, \ldots, A_n\}$ and there is a permutation of H', $\text{perm}(H') = \{B_1, \ldots, B_n\}$, such that $A_i = B_i$ for all $1 \leq i \leq n$.

Introduce
$< \{H\} \cup Gs, C_U, C_B > \longmapsto < Gs, \{H\} \cup C_U, C_B >$
if H is a user-defined constraint

Simplify
$< Gs, H' \cup C_U, C_B > \longmapsto < Gs \cup B, C_U, C_B >$
if $(H \texttt{ <=> } G \mid B) \in P$ and $C_B \rightarrow (H =_{set} H') \wedge answer(G)$

Propagate
$< Gs, H' \cup C_U, C_B > \longmapsto < Gs \cup B, H' \cup C_U, C_B >$
if $(H \texttt{ ==> } G \mid B) \in P$ and $C_B \rightarrow (H =_{set} H') \wedge answer(G)$

The constraint handling engine (CHE) applies CHRs to user-defined constraints in Gs and C_U whenever all user-defined constraints needed in the multi-head are present and the guard is satisfied. A guard G is *satisfied* if its local execution does not involve user-defined constraints and the result $answer(G)$ is entailed (implied) by the built-in constraint store C_B. Equality is entailed between two terms if they match. To *introduce* a user-defined constraint means to take it from the goal literals Gs and put it into the user-defined constraint store C_U. To *simplify* user-defined constraints H' means to replace them by B if H' matches the head H of a simplification CHR H `<=>` G `|` B and the guard G is satisfied. To *propagate from* user-defined constraints H' means to add B to Gs if H' matches the head H of a propagation CHR H `==>` G `|` B and G is satisfied.

Unfold
$< \{H'\} \cup Gs, C_U, C_B > \longmapsto < Gs \cup B, C_U, \{H = H'\} \cup C_B >$
if $(H$:- $B) \in P$.

The logic programming engine (LPE) unfolds goals in Gs. To *unfold* an atomic goal H' means to look for a clause H: – B and to replace the H' by $(H = H')$ and B. As there are usually several clauses for a goal, unfolding is nondeterministic and thus a goal can be solved in different ways using different clauses. There can be CLP clauses for user-defined constraints. Thus they can be unfolded as well. This unfolding is called *(built-in) labeling*. Details can be found in [Fru92, B*94].

Label
$< Gs, \{H'\} \cup C_U, C_B > \longmapsto < Gs \cup B, C_U, \{H = H'\} \cup C_B >$
if $(H$:- $B) \in P$ and $(label\_with\ H''\ if\ G) \in P$ and $C_B \rightarrow (H' = H'') \wedge answer(G)$

Note that any constraint solver written with CHRs will be *incremental* and *concurrent*. By "incremental" we mean that constraints can be added to the constraint store one at a time using the "introduce"-transition. Then CHRs may fire and simplify the user-defined constraint store. The rules can be applied concurrently, even using chaotic iteration (i.e. the same constraint can be simplified by different rules at the same time), because correct CHRs can only replace constraints by equivalent ones or add redundant constraints.

3 Examples

In the following we illustrate the behavior of Prolog extended with CHRs.

3.1 Booleans

In the domain of boolean constraints, the behavior of an and-gate may be described by rules such as

- **If** one input is 0 **then** the output is 0,
- **If** the output is 1 **then** both inputs are 1.

We can define the and-gate with constraint handling rules as:

```
and(X,Y,Z) <=> X=0 | Z=0.
and(X,Y,Z) <=> Y=0 | Z=0.
and(X,Y,Z) <=> Z=1 | X=1,Y=1.
and(X,Y,Z) <=> X=1 | Y=Z.
and(X,Y,Z) <=> Y=1 | X=Z.
and(X,Y,Z1),and(X,Y,Z2) ==> Z1=Z2.
```

The first rule says that the constraint goal `and(X,Y,Z)`, when it is known that the first input argument `X` is 0, can be reduced to asserting that the output

Z must be 0. Hence the query `and(X,Y,Z),X=0` will result in `X=0, Z=0`. The last rule says that if a goal contains both `and(X,Y,Z1)` and `and(X,Y,Z2)` then a consequence is that `Z1` and `Z2` must be the same.

Consider the well-known full-adder circuit:

```
add(I1,I2,I3,O1,O2):-
    xor(I1,I2,X1),
    and(I1,I2,A1),
    xor(X1,I3,O1),
    and(I3,X1,A2),
    or(A1,A2,O2).
```

The query `add(I1,I2,0,O1,1)` will produce `I1=1,I2=1,O1=0`. The computation proceeds as follows: Because `I3=0`, the result of the and-gate with input `I3`, the output `A2`, must be 0. As `O2=1` and `A2=0`, the other input `A1` of the or-gate must be 1. Because `A1` is also the output of an and-gate, its inputs `I1` and `I2` must be both 1. Hence the output `X1` of the first xor-gate must be 0, and therefore also the output `O1` of the second xor-gate must be 0.

3.2 Inequalities

In this example, we define an inequality constraint.

```
   % Constraint Declaration
(1a) constraints ≤/2.
(1b) label_with X≤Y if ground(X).
(1b) label_with X≤Y if ground(Y).

   % Constraint Definition
(2a) X≤Y :- leq(X,Y).
(2b) leq(0,Y).
(2c) leq(s(X),s(Y)) :- leq(X,Y).

   % Constraint Handling
(3a) X≤Y <=> X=Y | true.  % reflexivity
(3b) X≤Y,Y≤X <=> X=Y. % identity
(3c) X≤Y,Y≤Z ==> X≤Z. % transitivity
```

In clause (2a), $\leq$ is defined by a predicate `leq` which is defined by the two CLP clauses (2b) and (2c). The predicate definition specifies the user-defined constraint, it is thus called a *constraint definition*. The CHRs of (3) specify how $\leq$ simplifies and propagates as a constraint. They implement reflexivity, identity and transitivity of less-than-or-equal in a straightforward way. CHR (3a) states that `X≤X` is logically true. Hence, whenever we see the goal `X≤X` we can simplify it to `true`. Similarly, CHR (3b) means that if we find `X≤Y` as well as `X≤Y` in the current goal, we can replace it by the logically equivalent `X=Y`. CHR (3a) detects satisfiability of a constraint, and CHR (3b) solves a conjunction

of constraints returning a substitution. CHR (3c) states that the conjunction X≤Y,Y≤Z implies X≤Z.

If no simplification and propagation is possible anymore, a constraint goal is chosen for labeling and its constraint definition is executed. The label declaration (1b) and (1c) state that we may call X≤Y as a predicate if either X or Y are ground.

Some examples:

```
   :- A≤B, A=B.
% by CHR 3a
     true.

   :- A≤B, B≤A.
% by CHR 3b
     A=B.

   :- A≤B,C≤A,B≤C.
% C≤A,A≤B propagates C≤B by 3c.
% C≤B,B≤C simplifies to B=C by 3b.
% B≤A,A≤B simplifies to A=B by 3b.
     A=B,B=C.

   :- s(s(0))≤A,A≤s(s(s(0))).
% s(s(0))≤A,A≤s(s(s(0))) propagates s(s(0))≤s(s(s(0))).
% Labeling with s(s(0))≤s(s(s(0))) succeeds.
% Labeling with s(s(0))≤A succeeds with A=s(s(X)).
% Labeling with A≤s(s(s(0))) succeeds with X=0.
     A=s(s(0)).
% On backtracking A≤s(s(s(0))) succeeds with X=s(0).
     A=s(s(s(0))).
% On backtracking A≤s(s(s(0))) fails.
     false.
```

3.3 Implementation

The operational semantics are still far from the actual workings of an efficient implementation. At the moment, there exist two implementations, one prototype in LISP [Her93], and one fully developed compiler in a Prolog extension.

The compiler for CHRs together with a manual is available as a library of ECLiPSe [B*94], ECRC's advanced constraint logic programming platform, utilizing its delay-mechanism and built-in meta-predicates to create, inspect and manipulate delayed goals. All ECLiPSe documentation is available by anonymous ftp from ftp.ecrc.de, directory /pub/eclipse/doc. In such a sequential implementation, the transitions are tried in the textual order given above. To reflect the complexity of a program in the number of CHRs, at most two head

constraints are allowed in a rule[3]. This restriction also makes complexity for search of the head constraints of a CHR linear in the number of constraints on average (quadratic in the worst case) by using partitioning and indexing methods. Termination of a propagation CHR is achieved by never firing it a second time with the same pair of head constraints.

The CHRs library includes a debugger and a visual tracing toolkit as well as a full color demo using geometric constraints in a real-life application for wireless telecommunication. About 20 constraint solvers currently come with the release - for booleans, finite domains (similar to CHIP [VH89]), also over arbitrary ground terms, reals and pairs, incremental path consistency, temporal reasoning (quantitative and qualitative constraints over time points and intervals [Fru94]), for solving linear polynomials over the reals (similar to CLP(R) [J*92]) and rationals, for lists, sets, trees, terms and last but not least for terminological reasoning [FrHa95]. The average number of rules in a constraint solver is as low as 24. Typically it took only a few days to produce a reasonable prototype solver, since the usual formalisms to describe a constraint theory, i.e. inference rules, rewrite rules, sequents, first-order axioms, can be expressed as CHRs programs in a straightforward way. Thus one can directly express how constraints simplify and propagate without worrying about implementation details. Starting from this executable specification, the rules then can be refined and adapted to the specifics of the application.

On a wide range of solvers and examples, the run-time penalty for our declarative and high-level approach turned out to be a constant factor in comparison to dedicated built-in solvers (if available). Moreover, the slow-down is often within an order of magnitude. On some examples (e.g. those involving finite domains with the element-constraint), our approach is faster, since we can exactly define the amount of constraint simplification and propagation that is needed. This means that for performance and simplicity the solver can be kept as incomplete as the application allows it. Some solvers (e.g. disjunctive geometric constraints in the phone demo) would be very hard to recast in existing CLP languages.

4 Reasoning

Finally, a few words on reasoning about constraints defined by CHRs. Besides correctness, we can also prove *termination* and *confluence* of CHRs viewed as *conditional rewrite rules* by adopting and extending well-known techniques such as termination proofs and unfailing completion from rewriting systems. Briefly, termination is proved by giving an ordering on atoms showing that the body of a rule is always smaller than the head of the rule. Such an ordering in addition introduces an intuitive notion of a "simpler" constraint, so that we also support the intuition that constraints get indeed simplified. Note that when combining constraint solvers that share constraints, nonterminating simplification steps may arise even if each solver is terminating. E.g. one solver defines less-than in terms of greater-than and the other defines greater-than in terms of less-than.

[3] A rule with more head constraints can be rewritten into several two-headed rules.

The notion of confluence is important for combining constraint solvers as well as for concurrent applications of CHRs. Concurrent CHRs are not applied in a fixed order. As correct CHRs are logical consequences of the program, any result of a simplification step will have the same meaning, however it is not guaranteed anymore that the result is syntactically the same. In particular, a solver may be complete with one order of applications but incomplete with another one. Syntactically different constraint evaluations may also arise if combined solvers share constraints, depending on which solver comes first. A set of CHRs is *confluent* (or equivalently: *Church Rosser*), if each possible order of applications starting from any goal leads to the same resulting goal. An extension of unfailing completion makes a set of CHRs confluent.

5 Conclusions

Constraint handling rules (CHRs) are a language extension for implementing user-defined constraints. Rapid prototyping of novel applications for constraint techniques is encouraged by the high level of abstraction and declarative nature of CHRs.

References

[B*94] P. Brisset et al., ECLiPSe 3.4 Extensions User Manual, ECRC Munich, Germany, July 1994.

[Fru92] T. Frühwirth, *Constraint Simplification Rules*, Technical Report ECRC-92-18, ECRC Munich, Germany, July 1992 (revised version of Internal Report ECRC-LP-63, October 1991), available by anonymous ftp from ftp.ecrc.de, directory pub/ECRC_tech_reports/reports, file ECRC-92-18.ps.Z, Ongoing work presented at Workshop Logisches Programmieren, Goosen/Berlin, Germany, October 1991, Workshop on Rewriting and Constraints, Dagstuhl, Germany, October 1991, and JICSLP'92 Workshop on Constraint Logic Programming, Wasington D.C., USA, November 1992.

[F*92] T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy and M. Wallace. *Constraint Logic Programming - An Informal Introduction*, Chapter in Logic Programming in Action, Springer LNCS 636, September 1992 (also Technical Report ECRC-93-05, ECRC Munich, Germany, February 1993). Presented at Logic Programming Summer School, Zurich. Switzerland and at FGCS 92, Workshop on Constraint Logic Programming, Tokyo, Japan, June 1992.

[Fru93a] T. Frühwirth, *Entailment Simplification and Constraint Constructors for User-Defined Constraints*, Workshop on Constraint Logic Programming, Marseille, France, March 1993.

[Fru93b] T. Frühwirth, *User-Defined Constraint Handling*, Abstract, ICLP 93, Budapest, Hungary, MIT Press, June 1993.

[Fru94] T. Frühwirth, *Temporal Reasoning with Constraint Handling Rules*, Technical Report ECRC-94-05, ECRC Munich, Germany, February 1994 (first published as CORE-93-08, January 1993), available by anonymous ftp from ftp.ecrc.de, directory pub/ECRC_tech_reports/reports, file ECRC-94-05.ps.Z.

[FrHa95] T. Frühwirth and P. Hanschke, *Terminological Reasoning with Constraint Handling Rules*, Chapter in Principles and Practice of Constraint Programming (P. Van Hentenryck and V.J. Saraswat, Eds.), MIT Press, to appear (revised version of Technical Report ECRC-94-06, ECRC Munich, Germany, February 1994, available by anonymous ftp from ftp.ecrc.de, directory pub/ECRC_tech_reports/reports, file ECRC-94-06.ps.Z). Presented at Workshop on Principles and Practice of Constraint Programming, Newport, RI, USA, April 1993.

[Fru95] T. Frühwirth, *Constraint Handling Rules*, Chapter in "Constraints: Basics and Trends" (A. Podelski, ed.), Springer LNCS, to appear. Based on Tutorial at the Spring school on Constraints, Chatillon/Seine, France, May 94 and at the German Workshop on Logic Programming, Zurich, Switzerland, October 94.

[HaJa90] S. Haridi and S. Janson, Kernel Andorra Prolog and its Computation Model, Seventh International Conference on Logic Programming, MIT Press, 1990, pp. 31-46.

[Her93] Eine homogene Implementierungsebene fuer einen hybriden Wissensrepraesentationsformalismus, Master Thesis, in German, University of Kaiserslautern, Germany, April 1993.

[HS90] M. Höhfeld and G. Smolka, Definite Relations over Constraint Languages. LILOG Report 53, IBM Deutschland, West Germany, October 1988.

[J*92] J. Jaffar et al., The CLP(R) Language and System, ACM Transactions on programming Languages and Systems, Vol.14:3, July 1992, pp. 339-395.

[JaLa87] J. Jaffar and J.-L. Lassez, Constraint Logic Programming, ACM 14th POPL 87, Munich, Germany, January 1987, pp. 111-119.

[JaMa94] J. Jaffar and M. J. Maher, Constraint Logic Programming: A Survey, Journal of Logic Programming, 1994:19,20:503-581.

[Sar93] V. A. Saraswat, Concurrent Constraint Programming, MIT Press, Cambridge, 1993.

[Sha89] E. Shapiro, The Family of Concurrent Logic Programming Languages, ACM Computing Surveys, 21(3):413-510, September 1989.

[Smo91] G. Smolka, Residuation and Guarded Rules for Constraint Logic Programming, Digital Equipment Paris Research Laboratory Research Report, France, June 1991.

[VH89] P. Van Hentenryck, Constraint satisfaction in Logic Programming, MIT Press, Cambridge, 1989.

[VH91] P. van Hentenryck, Constraint Logic Programming, The Knowledge Engineering Review, Vol 6:3, 1991, pp 151-194.