Programming in Constraint Handling Rules

Thom Frühwirth

Faculty of Computer Science, University of Ulm, Germany
www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/

Background Material for Tutorial at CP2005, October 2005
Slides will be available on my homepage.
You will find papers related to CHR at the CHR 2005 workshop,
but also at CP, ICLP and some workshops.
You are also invited to the CHR webpages
http://www.cs.kuleuven.ac.be/ dtai/projects/CHR/

Abstract. Constraint Handling Rules (CHR) is a concurrent committed-choice constraint programming language, developed in the 1990s for the implementation of constraint solvers. It is traditionally an extension to other programming languages – especially constraint logic programming languages – but has been used increasingly as a general-purpose programming language in the recent past. With CHR, one can specify, implement and analyse algorithms in a concise and compact manner by executable inference rules.

1 Introduction

Constraint Handling Rules (CHR) [8, 13] is a concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multi-sets of constraints (atomic formulae) until no more change happens.

In CHR, one distinguishes two main kinds of rules: *Simplification rules* replace constraints by simpler constraints while preserving logical equivalence, e.g. $\mathtt{X}\geq\mathtt{Y} \wedge \mathtt{Y}\geq\mathtt{X} \Leftrightarrow \mathtt{X}=\mathtt{Y}$. *Propagation rules* add new constraints, which are logically redundant, but may cause further simplification, e.g. $\mathtt{X}\geq\mathtt{Y}\wedge\mathtt{Y}\geq\mathtt{Z} \Rightarrow \mathtt{X}\geq\mathtt{Z}$.

The combination of propagation and multi-set transformation of logical formulae in a rule-based language that is concurrent, guarded and constraint-based make CHR a rather unique and powerful declarative programming language. For example, any terminating and confluent CHR program will automatically implement a concurrent any-time (approximation) and on-line (incremental) algorithm. CHR tries to bridge the gap between theory and practice, between logical specification and executable program by abstraction and the concepts of computational logic. CHR supports rapid prototyping by giving the programmer efficiently executable specifications. Through the notion of constraint, CHR does not distinghuish between data and computation, it therefore naturally supports active data and suspensions/continuations in a high-level way. Multi-headed rules provide implicit iteration instead of cumbersome looping constructs.

CHR does not necessarily impose itself as a new programming language, but as a language extension that blends in with the syntax of its host language, be it Prolog, Lisp, Haskell or Java [23]. CHR is also available as WebCHR for online experimentation with more than 40 constraint solvers. The CHR webpages also link to 500+ papers mentioning CHR.

CHR was motivated by the inference rules that are traditionally used in computer science to define logical relationships and fixpoint computation in the most abstract way. CHR has many roots and combines their attractive features in a novel way. First of all, Prolog, constraint logic programming and concurrent committed-choice logic programming are direct ancestors of CHR. More concretely, it was influenced by the demons and forward rules of the CHIP language. It gratefully adapts concepts from term rewriting systems, but goes beyond them by providing propagation rules, logical variables, constraints and more. Like Automated Theorem Proving, it uses formulae to derive new information, but only in a restricted syntax and directional way that makes the difference between the art of proof search and an efficient programming language. Still, extending CHR with disjunction and existential quantifiers in $\mathrm{CHR}^{\vee}$ makes it quite suitable for theorem proving tasks. Other influences were the chemical abstract machine, and, of course, production rule systems in general, but also integrity constraints found in relational databases. We also would like to mention rewriting logic here, that was independently developed around the same time and that shares some motivations and ideas with CHR. Another related independent development were the event-condition-action rules that arose deductive database research.

If asked what distinguishes CHR from all these programming languages and computational systems, the quick answer is "propagation rules", then "multi-head/multi-set transformation", "constraints", reliance on logic, be it classical or not, and then one may add all the other characteristics of CHR.

One of the attractive features of the Constraint Handling Rules (CHR) programming language is its declarative semantics where rules are read as formulae in first-order predicate logic. The clean semantics of CHR facilitates non-trivial program analysis [25] and transformation. In particular, confluence analysis is an issue in CHR, since rule application is committed-choice, it is never undone. *Confluence* aks the question if a program produces the same result no matter which of the applicable rules are applied in which order. There is a decidable, sufficient and necessary criterion for confluence [5] that returns the problematic cases of rules applications that rule out each other.

Over time CHR has proven useful for many tasks outside its original field of application in constraint reasoning and computational logic[1], be it agent programming, multi-set rewriting or production rule systems. Recent applications of CHR range from type systems and time tabling to ray tracing and cancer diagnosis [4, 23]. In these applications, conjunctions of constraints are best regarded as interacting collections of concurrent agents or processes.

[1] Integrating deduction and abduction, bottom-up and top-down, forward and backward chaining, inegrity constraints and tabulation.

We give an overview of syntax and semantics for Constraint Handling Rules (CHR) [8, 13] as well as of results on confluence and completion [1, 5], and on operational equivalence [2, 3]. We then show how to program in CHR starting with some small examples, followed by constraint solvers for Booleans, arithmetic equations and the global lexicographic order constraint and finally give an optimal implementation of the classical union-find algorithm.

2 Syntax

Constraints are predicates of first-order logic. We use two disjoint sets of predicate symbols for two different kinds of constraints: *built-in (pre-defined) constraint symbols* which are solved by a given constraint solver, and *CHR (user-defined) constraint symbols* which are defined by the rules in a CHR program. Built-in constraints include $=$, *true*, and *false*. The semantics of the built-in constraints is defined by a consistent first-order *constraint theory CT*. In particular, *CT* defines $=$ as the syntactic equality over finite terms.

Definition 1. A *CHR program* is a finite set of rules. There are three kinds of rules:

$$\begin{array}{ll} \textit{Simplification rule:} & \textit{Name} \ @ \ H \Leftrightarrow C \mid B, \\ \textit{Propagation rule:} & \textit{Name} \ @ \ H \Rightarrow C \mid B, \\ \textit{Simpagation rule:} & \textit{Name} \ @ \ H \setminus H' \Leftrightarrow C \mid B, \end{array}$$

where *Name* is an optional, unique identifier of a rule, the *head* H, H' is a non-empty comma-separated conjunction of CHR constraints, the *guard* C is a conjunction of built-in constraints, and the *body* B is a goal. A *goal (query, problem)* is a conjunction of built-in and CHR constraints. The empty conjunction is denoted by the built-in constraint *true*. A trivial guard expression "`true |`" can be omitted from a rule. A CHR constraint symbol is *defined* in a CHR program if it occurs in the head of a rule in the program.

Simpagation rules abbreviate simplification rules of the form

$$\textit{Name} \ @ \ H \wedge H' \Leftrightarrow C \mid H \wedge B.$$

Therefore they are not treated separately. But note that they can be written and implemented more efficiently than their simplification rule correspondences.

Example 1. We define a CHR constraint for a partial order relation $\leq$:

```
r1 @ X≤X ⇔ true.
r2 @ X≤Y ∧ Y≤X ⇔ X=Y.
r3 @ X≤Y ∧ Y≤Z ⇒ X≤Z.
r4 @ X≤Y ∧ X≤Y ⇔ X≤Y.
```

The CHR program implements reflexivity (`r1`), antisymmetry (`r2`), transitivity (`r3`) and redundancy (`r4`) in a straightforward way. The reflexivity rule `r1` states that `X`$\leq$`X` is logically true. The antisymmetry rule `r2` means `X`$\leq$`Y` $\wedge$ `Y`$\leq$`X` is logically equivalent to `X=Y`. The transitivity rule `r3` states that the conjunction of `X`$\leq$`Y` and `Y`$\leq$`Z` implies `X`$\leq$`Z`. The redundancy rule `r4` states that `X`$\leq$`Y` $\wedge$ `X`$\leq$`Y` is logically equivalent to `X`$\leq$`Y`.

3 Operational Semantics

At runtime, a CHR program is provided with an initial state and will be executed until either no more rules are applicable or a contradiction occurs.

The operational semantics of CHR is given by a transition system (Fig. 1). Let P be a CHR program. We define the transition relation $\mapsto$ by introducing two computation steps (transitions), one for each kind of CHR rule. *States* are goals, i.e. conjunctions of built-in and CHR constraints. States are also called *(constraint) stores*. In the figure, all upper case letters are meta-variables that stand for conjunctions of constraints. CT is the constraint theory for the built-in constraints. G_{bi} denotes the built-in constraints of G, which is part of the current state/goal.

Simplify

If	$(H \Leftrightarrow C \mid B)$ is a fresh variant with variables $\bar{x}$ of a rule in P
and	$CT \models \forall\, (G_{bi} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G) \mapsto_P^{\textbf{Simplify}} (B \wedge G \wedge H{=}H' \wedge C)$

Propagate

If	$(H \Rightarrow C \mid B)$ is a fresh variant with variables $\bar{x}$ of a rule in P
and	$CT \models \forall\, (G_{bi} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G) \mapsto_P^{\textbf{Propagate}} (H' \wedge B \wedge G \wedge H{=}H' \wedge C)$

Fig. 1. Computation Steps of Constraint Handling Rules

CHR rules are applied exhaustively, until a fixed-point is reached, to the initial state. A simplification rule $H \Leftrightarrow C \mid B$ *replaces* instances of the CHR constraints H by B provided the guard C holds. A propagation rule $H \Rightarrow C \mid B$ instead *adds* B to H. If new constraints arrive, rule applications are restarted.

Trivial non-termination of the **Propagate** computation step is avoided by applying a propagation rule at most once to the same constraints. A more concrete operational semantics that reflects this behavior is presented in [1].

A rule is *applicable*, if its head constraints are matched by constraints in the current goal one-by-one and if, under this matching, the guard of the rule is implied by the built-in constraints in the goal. Any of the applicable rules can be applied, and the application cannot be undone, it is committed-choice.

In more detail, a fresh variant of a rule is *applicable to a state $H' \wedge G$* if H' matches its head H and if its guard C is implied by the built-in constraints appearing in G. A *fresh variant* of a rule is obtained by renaming its variables to different fresh variables, listed in the sequence $\bar{x}$. *Matching* (one-sided unification) succeeds if H' is an instance of H, i.e. it is only allowed to instantiate (bind) variables of H but not variables of H'. Matching is logically expressed by equating H' and H but existentially quantifying all variables from the rule, $\bar{x}$. This equation $H'{=}H$ is shorthand for pairwise equating the arguments of the constraints in H' and H, provided their constraint symbols are equal. Note that conjuncts can be permuted.

We usually will drop the reference to the type of rule and to the program P for the symbol $\mapsto$. A *computation (derivation)* of a goal G is a sequence $S_0, S_1, \ldots$ of states with $S_i \mapsto S_{i+1}$ beginning with the initial state $S_0 = G$ and ending in a final state or not terminating. An *initial state* for a goal G is the state G. A *final state* is one where either no computation step is possible anymore or where the built-in constraints are inconsistent. The notation $\mapsto^*$ denotes the reflexive and transitive closure of $\mapsto$.

Example 2. Recall the solver program for $\leq$ of Example 1. Operationally the rule `r1` removes occurrences of constraints that match `X≤X`. The antisymmetry rule `r2` means that if we find `X≤Y` as well as `Y≤X` in the current goal, we can replace them by the logically equivalent `X=Y`. The transitivity rule `r3` propagates constraints. It adds the logical consequence `X≤Z` as a redundant constraint, but does not remove any constraints. The redundancy rule `r4` absorbs multiple occurrences of the same constraint.

A computation of the goal `A≤B ∧ C≤A ∧ B≤C` proceeds as follows:

$$
\begin{array}{ll}
\underline{\mathtt{A}\leq\mathtt{B}} \wedge \underline{\mathtt{C}\leq\mathtt{A}} \wedge \mathtt{B}\leq\mathtt{C} & \mapsto^{\textbf{Propagate}} \\
\mathtt{A}\leq\mathtt{B} \wedge \mathtt{C}\leq\mathtt{A} \wedge \underline{\mathtt{B}\leq\mathtt{C}} \wedge \underline{\mathtt{C}\leq\mathtt{B}} & \mapsto^{\textbf{Simplify}} \\
\underline{\mathtt{A}\leq\mathtt{B}} \wedge \underline{\mathtt{B}\leq\mathtt{A}} \wedge \mathtt{B}{=}\mathtt{C} & \mapsto^{\textbf{Simplify}} \\
\mathtt{A}{=}\mathtt{B} \wedge \mathtt{B}{=}\mathtt{C} &
\end{array}
$$

Starting from a circular relationship, we have found out that the three variables must be the same.

Refined Semantics

This high-level description of the operational semantics of CHR leaves two main sources of non-determinism: the order in which constraints of a query are processed and the order in which rules are applied. As in Prolog, almost all CHR implementations execute queries from left to right and apply rules top-down in the textual order of the program. This behavior has been formalized in the so-called *refined semantics* that was proven to be a concretization of the standard operational semantics [7].

In this refined semantics that is closer to that of most implementations, a CHR constraint in a query can be understood as a procedure that goes efficiently through the rules of the program in the order they are written. We consider such a constraint to be *active*. When it matches a head constraint of a rule, it will look for the other, *partner constraints* of the head in the *constraint store* and check the guard until an applicable rule is found. If the active constraint has not been removed after trying all rules, it will be put into the constraint store. Constraints from the store will be reconsidered (woken) if newly added built-in constraints constrain variables of the constraint, because then rules may become applicable since their guards are now implied.

Parallelism

In [12] a general *parallel* execution model for CHR is presented. It relies on a monotonicity property (applicable rules cannot become in-applicable during a computation). Intuitively, in a parallel execution of a CHR program, rules can be applied to separate parts of the problem in parallel without interference. Analogous concurrency constructions were suggested for other (constraint) logic programming languages, e.g. [21]. But in CHR, more parallism is possible: rules can be applied to overlapping parts of a query if at most one of the rules removes the overlap. This relates parallism to confluence [12].

4 Declarative Semantics

Owing to the tradition of logic and constraint logic programming, CHR features – besides a well-defined operational semantics – a *declarative semantics*, i.e. a direct translation of a CHR program into a first-order logical formula. In the case of constraint handlers, this is a useful tool, since it strongly facilitates proofs of a program's faithful handling of constraints.

The *logical reading (meaning)* of a simplification rule

$$H \Leftrightarrow C \mid B,$$

is a logical equivalence provided the guard holds,

$$\forall (C \rightarrow (H \leftrightarrow \exists \bar{y}\ B)),$$

where $\forall F$ denotes the universal closure of a formula F and $\bar{y}$ are the variables that appear only in the body B.

The logical reading of a propagation rule

$$H \Rightarrow C \mid B,$$

is an implication provided the guard holds

$$\forall (C \rightarrow (H \rightarrow \exists \bar{y}\ B)).$$

The logical reading $\mathcal{P}$ of a CHR program P is the conjunction of the logical readings of its rules united with a the constraint theory CT that defines the built-in constraint symbols.

Soundness and Completeness

The following theorems show that the operational and the declarative first-order-logic semantics are strongly related.

Definition 2. *A* computable constraint *of a state* S_0 *is the logical reading* S'_a *of a derived state of* S_0*. An* answer (constraint) *of a state* S_0 *is the logical reading* S'_n *of a final state of a computation from* S_0*.*

The following theorems are proved in [5]:

Theorem 1. (Soundness). *Let P be a CHR program and* S_0 *be an initial state. If* S_0 *has a computation with answer constraint* S'_n*, then* $P' \cup CT \models \forall(S'_0 \leftrightarrow S'_n)$*.*

Theorem 2. (Completeness). *Let P be a CHR program and* S_0 *be an initial state with at least one finite computation. If* $P' \cup CT \models \forall(S'_0 \leftrightarrow S'_n)$*, then* S_0 *has a computation with answer constraint* S'_m *such that* $P' \cup CT \models \forall(S'_m \leftrightarrow S'_n)$*.*

Linear-Logic Semantics

The classical-logic declarative semantics, however, poses a problem, when applied to non-traditional uses of CHR, i.e. CHR programs that use CHR as a general-purpose concurrent programming language. Many implemented algorithms do not have a first-order classical logical reading, especially when these algorithms are deliberately non-confluent. This may lead to logical readings which are inconsistent with the intended meaning. This problem has recently been demonstrated in [12, 24] and constitutes the motivation for the development of an alternative declarative semantics. It is based on a subset of *linear logic* [15], which can model resource consumption and therefore more accurately describe the operational behavior of simplification rules [6]. In the cited thesis, soundness and a rather strong completeness theorem are proven.

5 Confluence and Completion

The confluence property of a program guarantees that any computation for a goal results in the same final state no matter which of the applicable rules are applied.

Definition 3. A CHR program is *confluent* if for all states S, S_1, S_2: If $S \mapsto^* S_1$ and $S \mapsto^* S_2$ then the pair of states (S_1, S_2) is joinable.

A pair of states (S_1, S_2) is *joinable* if there exist states T_1 and T_2 such that $S_1 \mapsto^* T_1$ and $S_2 \mapsto^* T_2$ where T_1 and T_2 are identical up to renaming of local variables and logical equivalence of built-in constraints.

Local variables are those that are introduced during a computation, but do not occur in the initial state.

To analyze confluence of a given CHR program we cannot check joinability starting from any given ancestor state S, because in general there are infinitely many such states. However for terminating programs, one can restrict the joinability test to a finite number of "minimal" states, the so-called critical states as explained below.

Definition 4. Let R_1 be a simplification rule and R_2 be a (not necessarily different) rule, whose variables have been renamed apart. Let $H_i \wedge A_i$ be the head and C_i be the guard of rule R_i $(i = 1, 2)$. Then a *critical (ancestor) state of* R_1 *and* R_2 is

$$(H_1 \wedge A_1 \wedge H_2 \wedge (A_1{=}A_2) \wedge C_1 \wedge C_2),$$

provided A_1 and A_2 are non-empty conjunctions and $CT \models \exists((A_1{=}A_2) \wedge C_1 \wedge C_2)$.

Let S be a critical ancestor state of R_1 and R_2. If $S \mapsto S_1$ using rule R_1 and $S \mapsto S_2$ using rule R_2 then the tuple (S_1, S_2) is a *critical pair* of R_1 and R_2.

The following theorem from [1, 5] gives a decidable, sufficient and necessary condition for confluence of a terminating CHR program.

A CHR program is called *terminating*, if there are no infinite computations. For many existing CHR programs simple well-founded orderings are sufficient to prove termination [9, 10]. In general, such orderings are not sufficient because of non-trivial interactions between simplification and propagation rules.

Theorem 3. A terminating CHR program is confluent iff all its critical pairs are joinable.

Example 3. Recall the program for $\leq$ of Example 1. Consider a critical ancestor state of `r2` and `r3` where $A_1 = A_2 =$ $\mathtt{X}{\leq}\mathtt{Y}$. This critical state is $\mathtt{X}{\leq}\mathtt{Y} \wedge \mathtt{Y}{\leq}\mathtt{X} \wedge \mathtt{Y}{\leq}\mathtt{Z}$ and gives raise to the following critical pair

$$(S_1, S_2) = (\mathtt{X}{=}\mathtt{Y} \ \wedge \ \mathtt{X}{\leq}\mathtt{Z}, \quad \mathtt{X}{\leq}\mathtt{Y} \ \wedge \ \mathtt{Y}{\leq}\mathtt{X} \ \wedge \ \mathtt{Y}{\leq}\mathtt{Z} \ \wedge \ \mathtt{X}{\leq}\mathtt{Z})$$

It is joinable: S_1 is a final state, i.e. no further computation step is possible. A computation beginning with S_2 results in S_1:

$$\begin{array}{ll}
\underline{\mathtt{X}{\leq}\mathtt{Y}} \ \wedge \ \underline{\mathtt{Y}{\leq}\mathtt{X}} \ \wedge \ \mathtt{Y}{\leq}\mathtt{Z} \ \wedge \ \mathtt{X}{\leq}\mathtt{Z} & \mapsto^{\mathbf{Simplify}} \\
\underline{\mathtt{X}{\leq}\mathtt{Z}} \ \wedge \ \underline{\mathtt{X}{\leq}\mathtt{Z}} \ \wedge \ \mathtt{X}{=}\mathtt{Y} & \mapsto^{\mathbf{Simplify}} \\
\mathtt{X}{\leq}\mathtt{Z} \ \wedge \ \mathtt{X}{=}\mathtt{Y} &
\end{array}$$

Completion

Completion is the process of adding rules to a non-confluent program until it becomes confluent. Rules are built from a non-joinable critical pair to allow a transition from one of the states into the other while maintaining termination. In contrast to other completion methods, in CHR we need in general more than one rule to make a critical pair joinable: a simplification rule and a propagation rule [2]. When these rules are added, new critical pairs may be produced, but also old non-joinable critical pairs may be removed, because the new rules make them joinable. Completion tries to continue introducing rules this way until the program becomes confluent. The essential part of a completion algorithm is the introduction of rules from critical pairs.

Definition 5. Let $\gg$ be a termination order and let $(C_{ud1} \land C_{bi1}\ ,\ C_{ud2} \land C_{bi2})$ be a critical pair, where the states are ordered such that C_{ud1} is a non-empty conjunction and $C_{ud1} \gg C_{ud2}$. Then the *orientation* of the critical pair results in the rules:

$$\begin{array}{l} C_{ud1} \Leftrightarrow C_{bi1} \mid C_{ud2} \land C_{bi2} \\ C_{ud2} \Rightarrow C_{bi2} \mid C_{bi1} \end{array}$$

The second rule is generated if C_{ud2} is a non-empty conjunction and $CT \not\models C_{bi2} \rightarrow C_{bi1}$.

Example 4.

In [2] it was shown that if the completion procedure stops successfully, then the resulting program is confluent. But completion cannot always be successful: completion is aborted if a critical pair cannot be transformed into rules (e.g. states cannot be ordered, or consist of different built-in constraints only). Completion may not terminate, this is the case when new rules produce new critical pairs, which require again new rules, and so on.

6 Operational Equivalence

A fundamental and hard question in programming language semantics is when two programs should be considered equivalent. For example correctness of program transformation can be studied only with respect to a notion of equivalence. Also, if modules or libraries with similar functionality are used together, one may be interested in finding out if program parts in different modules or libraries are equivalent. In the context of CHR, this case arises frequently when constraint solvers written in CHR are combined. Typically, a constraint is only partially defined in a constraint solver. We want to make sure that the operational semantics of the common constraints of two programs do not differ, and we are interested in finding out if they are equivalent.

The following definition clarifies when two programs are operationally equivalent: if for each goal, all final states in one program are the same as the final states in the other program.

Definition 6. Let P_1 and P_2 be programs. P_1 and P_2 are *operationally equivalent* if all states are P_1, P_2-joinable.

A state S is P_1, P_2*-joinable*, iff there are two computations $S \mapsto^*_{P_1} S_1$ and $S \mapsto^*_{P_2} S_2$, where S_1 and S_2 are final states, and S_1 and S_2 are identical up to renaming of local variables and logical equivalence of built-in constraints.

In [3], the authors gave a decidable, sufficient and necessary syntactic condition for operational equivalence of terminating and confluent CHR programs[2]: when testing operational equivalence, similar to our confluence test, we can restrict ourselves to a finite number of *minimal states* that consist of the head and the guard of a rule. These minimal states are run in both programs, and their outcome must be the same.

Definition 7. Let P_1 and P_2 be programs. Then *a minimal state of* P_1 *and* P_2 is defined as follows:

$$H \wedge C \text{ where } (H \ \odot \ C \mid B) \in P_1 \cup P_2 \text{ and } \odot \in \{ \Leftrightarrow , \Rightarrow \}$$

Theorem 4. Two terminating and confluent programs P_1 and P_2 are operationally equivalent iff all minimal states of P_1 and P_2 are P_1, P_2-joinable.

An example for operational equivalence checking is in the next section.

7 Small Programs

In the following we will use the concrete ASCII syntax of CHR implementations in Prolog (unless otherwise noted). Let $\leq$ and $<$ be built-in constraints now.

7.1 Minimum Constraint

We define `min` as CHR constraint, where `min(X,Y,Z)` means that `Z` is the minimum of `X` and `Y`:

```
r1 @ min(X,Y,Z) <=> X≤Y | Z=X.
r2 @ min(X,Y,Z) <=> Y≤X | Z=Y.
r3 @ min(X,Y,Z) <=> Z<X | Y=Z.
r4 @ min(X,Y,Z) <=> Z<Y | X=Z.
r5 @ min(X,Y,Z) ==> Z≤X, Z≤Y.
```

It can be shown that the program is terminating and confluent. The first two rules ($r1$) and ($r2$) correspond to the usual definition of `min`.

But we also want to be able to compute backwards. In CHR this is achieved by committed-choice rules that explicitely express the cases where a simplification is possible. So the two rules ($r3$) and ($r4$) simplify `min` if the order between the result `Z` and one of the input variables is known.

The last rule ($r5$) propagates constraints. It states that `min(X,Y,Z)` unconditionally implies `Z≤X`, `Z≤Y`. Operationally, CHR adds these logical consequences as redundant constraints and the `min` constraint is kept.

To the goal `min(1,2,M)` the first rule is applicable resulting in `M=1`.

To the goal `min(A,B,M)`, `A≥B` the second rule is applicable resulting in `M=B`, $\mathtt{A} \geq \mathtt{B}$.

[2] To the best of our knowledge, CHR is the only programming language in practical use that admits decidable operational equivalence

The goal `min(A,A,M)` leads to `A=M` via rule ($r1$). Alternatively, rule ($r2$) is applicable with the same result. However, since CHR is a committed-choice language, only one of the rules will be applied.

The goal `min(A,2,1)` leads to `A=1` via rule ($r4$).

The goal `min(A,2,3)` leads to failure via rule ($r5$).

Redundancy from a propagation rule is useful, as the goal `min(A,2,2)` shows. To this goal only the propagation rule is applicable, but to the resulting state the second rule becomes applicable:

$$
\begin{array}{ll}
 & \mathtt{min}(\mathtt{A},2,2) \\
\mapsto_{\mathbf{Propagate}\ (r5)} & \mathtt{min}(\mathtt{A},2,2), 2 \leq \mathtt{A}, 2 \leq 2 \\
\equiv & \mathtt{min}(\mathtt{A},2,2), 2 \leq \mathtt{A} \\
\mapsto_{\mathbf{Simplify}\ (r2)} & 2{=}2, 2 \leq \mathtt{A} \\
\equiv & 2 \leq \mathtt{A}
\end{array}
$$

In this way, we find out that for `min(A,2,2)` to hold, $2 \leq \mathtt{A}$ must hold.

Another interesting derivation involving the propagation rule is the following:

$$
\begin{array}{ll}
 & \mathtt{min}(\mathtt{A},\mathtt{B},\mathtt{M}), \mathtt{A}{=}\mathtt{M} \\
\mapsto_{\mathbf{Propagate}\ (r5)} & \mathtt{min}(\mathtt{A},\mathtt{B},\mathtt{M}), \mathtt{M} \leq \mathtt{A}, \mathtt{M} \leq \mathtt{B}, \mathtt{A}{=}\mathtt{M} \\
\equiv & \mathtt{min}(\mathtt{A},\mathtt{B},\mathtt{M}), \mathtt{M} \leq \mathtt{B}, \mathtt{A}{=}\mathtt{M} \\
\mapsto_{\mathbf{Simplify}\ (r1)} & \mathtt{M} \leq \mathtt{B}, \mathtt{A}{=}\mathtt{M}
\end{array}
$$

Operational Equivalence We would like to know if the following two CHR rules defining the user-defined constraint `min`

```
min(X,Y,Z) <=> X<Y | Z=X.
min(X,Y,Z) <=> X≥Y | Z=Y.
```

are operationally equivalent with these two rules

```
min(X,Y,Z) <=> X≤Y | Z=X.
min(X,Y,Z) <=> X>Y | Z=Y.
```

or if the union of the rules results in a better constraint solver for `min`.

Already the first minimal state, `min(X,Y,Z) <=> X<Y`, shows that the two programs are not operationally equivalent, since it can reduce with the first rule of the first program, but is a final state for the second program.

7.2 Fibonacci Numbers

The following terminating and confluent program computes Fibonacci numbers in the obvious way, that is, `fib(N,M)` is true if `M` is the `N`-th Fibonacci number.

```
fib(0,M) <=> M=1.
fib(1,M) <=> M=1.
fib(N,M) <=> N>1 |
    N1 is N-1, fib(N-1,M1), N2 is N-2, fib(N-2,M2), M is M1+M2.
```

The Prolog predicate `is` computes the value of the arithmetic expression on the right and equates it with the left argument.

As is well-known, the two recursive calls compute the same Fibonacci numbers again and again. We would like to reuse results of previous computations by using memoization (tabulation). It suffices to keep the `fib` constraints by turning the simplification rules above into propagation rules and to add a rule in front that expresses the functional dependency between the first and second argument of `fib`. By this simpagation rule, two computations (ongoing or finished) for the same Fibonacci number will be merged into one.

```
fib(N,M1) \ fib(N,M2) <=> M1=M2.
fib(0,M) ==> M=1.
fib(1,M) ==> M=1.
fib(N,M) ==> N>1 |
    N1 is N-1, fib(N-1,M1), N2 is N-2, fib(N-2,M2), M is M1+M2.
```

7.3 Primes Sieve

We implement the Sieve of Eratosthenes to compute primes by the following terminating and confluent[3] CHR program:

```
candidates(N) <=> N>1 | M is N-1,prime(N),candidates(M).
candidates(1) <=> true.

sift @ prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

The CHR constraint `candidates(N)` generates candidates for prime numbers, `prime(M)`, where `M` is from `2` to `N`. The candidates `prime` react with each other such that each number absorbs multiples of itself. In the end, only prime numbers remain. The essential multi-headed simpagation rule named `sift` works as follows: If there is a constraint `prime(I)` and some other constraint `prime(J)` such that `J mod I =:= 0` holds, i.e. `J` is a multiple of `I`, then keep `prime(I)` but remove `prime(J)`.

This example illustrates the use of multi-headed rules instead of explicit loops for iteration over data. In comparison, the typical Prolog program for the Primes Sieve is much more contrived, because one explicitely has to maintain a list of prime candidates.

```
primes(N,Ps) :- candidates(2,N,Ns), sift(Ns,Ps).

candidates(F,T,[]) :- F > T.
candidates(F,T,[F|Ns1]) :- F =< T, F1 is F+1, candidates(F1,T,Ns1).

sift([],[]).
```

[3] Consider e.g. the critical state `prime(I),prime(J),prime(K), J mod I =:= 0, K mod J =:= 0`.

```
sift([P|Ns],[P|Ps1]) :- filter(Ns,P,Ns1), sift(Ns1,Ps1).

filter([],P,[]).
filter([X|In],P,Out) :- 0 =:= X mod P, filter(In,P,Out).
filter([X|In],P,[X|Out1]) :- 0 =\= X mod P, filter(In,P,Out1).
```

Consider the declarative semantics of the CHR constraint `prime` as defined by the `absorb` rule:

$$\forall(M \bmod N = 0 \rightarrow (prime(M) \wedge prime(N) \leftrightarrow prime(N))$$

What this logical expression actually says is that "a number is prime, if it is a multiple of another prime number". The problem is that the `prime` constraints form an initially a range of integers representing candidates for primes. Only upon completion of the computation (that can be regarded as normalization) they do represent the actual primes. Predicate logic has no straightforward means to express this dynamics, but linear logic does [6].

CHR in Java The following code implement the same rules in JCHR, the first CHR implementation in Java, which is part of the Java Constraint Kit JCK [20]. The syntax of CHR rules has chosen to be similar to that of the host language Java.

```
handler primes {
class java.lang.Integer;
class IntUtil;

constraint prime(java.lang.Integer);
constraint candidates(java.lang.Integer);

rules { variable java.lang.Integer N, M, I, J;

{candidates(1)} <=> {true} ;

if (IntUtil.gt(N, 1)) {candidates(N)} <=>
         {M=IntUtil.dec(N) && prime(N) && candidates(M)};

if(IntUtil.modNull(J, I)) {prime(I) &\& prime(J)} <=> {true} sift;
     }
}
```

A more recent implementation of CHR in Java, K.U.Leuven JCHR system [19], uses the more traditional Prolog-style syntax of CHR.

```
import util.arithmetics.primitives.intUtil;

handler primes {
```

```
    constraint candidates(int);
    constraint prime(int);

    rules { variable int N, X, Y;

      candidates(1) <=> true.
      candidates(N) <=> prime(N), candidates(intUtil.dec(N)).

      sift @ prime(Y) \ prime(X) <=> intUtil.modZero(X, Y) | true.
    }
}
```

There are at least two more implementations of CHR in Java [23].

8 Constraint Solvers

8.1 Boolean Constraint Solver

Boolean (propositional logic) constraints can be solved by different techniques [18]. In the following terminating and confluent[4] Boolean constraint solver [13] a local consistency algorithm is used. It simplifies one atomic Boolean constraint at a time into one or more syntactic equalities whenever possible. The rules for $X \sqcap Y = Z$, which is represented in relational form as `and(X,Y,Z)`, are as follows. For the other connectives, they are analogous.

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

For example, the first rule says that the constraint `and(X,Y,Z)`, when it is known that the input `X` is `0`, can be reduced to asserting that the output `Z` must be `0`. Hence, the constraint `and(X,Y,Z), X=0` will result in `X=0, Z=0`. Note that a rule for `Z=0` is missing, since this case admits no simplification into syntactic equalities.

The above rules are based on the idea that, given a value for one of the variables in a constraint, we try to detect values for other variables. However, the Boolean solver goes beyond propagating values, since it also propagates equalities between variables. For example, `and(1,Y,Z), neg(Y,Z)` will reduce to `false`, and this cannot be achieved by value propagation alone.

[4] Consider e.g. the critical pair `and(0,Y,1)`.

8.2 Linear Polynomial Equation Solving

Typically, in arithmetic constraint solvers, incremental variants of classical variable elimination algorithms [16] like Gaussian elimination for equations and Dantzig's Simplex algorithm for equations and inequations are implemented.

To illustrate the principle of *variable elimination*, we first consider equations only. A conjunction of equations is *in solved form* if the left-most variable of each equation does not appear in any other equation. We compute the solved form by eliminating multiple occurrences of variables.

- Choose an equation $a_1 * X_1 + \ldots + a_n * X_n + b = 0$.
- Make its left-most variable explicit: $X_1 = -(a_2 * X_2 + \ldots + a_n * X_n + b)/a_1$.
- Replace all other occurrences of X_1 by $-(a_2 * X_2 + \ldots + a_n * X_n + b)/a_1$.
- Normalize the resulting equations into allowed constraints (this is always possible).
- Repeat until in solved form.

In this solved form, all determined variables (those that take a unique value) are discovered.

Since constraints are typically processed incrementally in a constraint solver, we do not eliminate a variable in *all* other equations at once, but rather consider the other equations one by one. Also, we do not need to make a variable explicit, but keep the original equation. Two rules suffice for the solver [13]:

```
eliminate @ A1*X+P1 eq 0, PX eq 0 <=>
            find(A2*X,PX,P2) |
            normalize(A2*(-P1/A1)+P2,P3),
            A1*X+P1 eq 0, P3 eq 0.

empty @ B eq 0 <=> number(B) | zero(B).
```

The `eliminate` rule performs variable elimination. It takes any pair of equations with a common occurrence of a variable, `X`. In the first equation, the variable appears left-most. This equation is used to eliminate the occurrence of the variable in the second equation. The first equation is left unchanged.

In the guard, the built-in `find(A2*X,PX,P2)` tries to find the expression `A2*X` in the polynom `PX`, where `X` is the common variable. The polynom `P2` is `PX` with `A2*X` removed. The built-in `normalize(E,P)` normalizes an arithmetic expression `E` into a linear polynomial `P`.

The `empty` rule says that if the polynomial contains no more variables, then the number `B` must be zero.

The solver is not confluent due to the `eliminate` rule, but produces the solved form. (If a set of equations is not in solved form, then one of the rules of the solver is applicable.)

Example 5. The two equations

```
1*X+3*Y+5 eq 0, 3*X+2*Y+8 eq 0
```

match the `eliminate` rule, the variable `X` in the second equation is removed with the help of

```
normalize(3*(-(3*Y+5)/1) + (2*Y+8), P3)
```

that computes `P3` to be `-7*Y+-7`. The resulting equations are

```
1*X+3*Y+5 eq 0, -7*Y+-7= 0
```

The `eliminate` rule is now applicable to the equations in reversed order, i.e. `Y` is removed from the first equation with the help of

```
normalize(3*(-(-7)/-7) + (1*X+5), P3)
```

The final result is:

```
1*X+2 eq 0,     -7*Y+-7= 0
```

So `X` is determined to be `-2` and `Y` is `-1`.

The solver can be extended by a rule to detect such determined variables:

```
determine @ A*X+B eq 0 <=> number(B) | X is -B/A.
```

The built-in `V is E` computes the result of the arithmetic expression `E` and equates it with the variable `V`.

8.3 Lexicographic Order Global Constraint

In [11], we give an executable specification of the global constraint[5] of lexicographic order in CHR. In contrast to previous approaches, the implementation is short and concise without giving up on linear time worst case time complexity. It is incremental and concurrent by nature of CHR. It is provably correct and confluent. It is independent of the underlying constraint system, and therefore not restricted to finite domains. In [11], we also show completeness of constraint propagation, i.e. that all possible consequences of the constraint are generated by the implementation.

A lexicographic order allows to compare sequences by comparing the elements of the sequences proceeding from start to end.

Definition 8. Given two sequences l_1 and l_2 of variables of the same length n, $[x_1, \ldots, x_n]$ and $[y_1, \ldots, y_n]$, then $l_1 \preceq_{lex} l_2$ iff n=0 or $x_1 < y_1$ or $x_1 = y_1$ and $[x_2, \ldots, x_n] \preceq_{lex} [y_2, \ldots, y_n]$.

This definition gives rise to the following logical specification:

$$\begin{aligned} l_1 \preceq_{lex} l_2 \leftrightarrow\ & (l_1=[] \wedge l_2=[]) \vee \\ & (l_1=[x|l_1'] \wedge l_2=[y|l_2'] \wedge x<y) \vee \\ & (l_1=[x|l_1'] \wedge l_2=[y|l_2'] \wedge x=y \wedge l_1' \preceq_{lex} l_2') \end{aligned}$$

[5] A global constraint admits an arbitrary number of variables as argument.

A First Implemention in CHR We now rewrite this logical specification into CHR rules for the lexicographic order constraint. We assume that the lists to be compared are given, while their elements are variables or constants. Since the three disjuncts of the specification are mutually exclusive, we can easily turn each clause into a CHR simplification rule where the guards ensure the mutual exclusion.

```
l1 @ [] lex [] <=> true.
l2 @ [X|L1] lex [Y|L2] <=> X<Y | true.
l3 @ [X|L1] lex [Y|L2] <=> X=Y | L1 lex L2.
```

These rules will apply when the lists are empty or when the relationship between the leading list elements `X` and `Y` is sufficiently known. The built-in constraints `X<Y` and `X=Y` are in the guards, so they check if the appropriate relationship between the variables holds. For example, the queries `[1] lex [2]`, `[X] lex [X]` and `[X] lex [Y], X<Y` will all reduce to `true`. To the queries `[X] lex [Y]`, `[X] lex [Y], X>=Y` and `[X] lex [Y], X>Y` no rules are applicable.

Adding Constraint Propagation While the above program is correct, the rules do not propagate any constraints except the trivial `true`. We must do better than that. We can derive a common consequence of the last two clauses,

$$(l_1{=}[x|l_1'] \wedge l_2{=}[y|l_2'] \wedge x{<}y) \;\vee\; (l_1{=}[x|l_1'] \wedge l_2{=}[y|l_2'] \wedge x{=}y \wedge l_1'{\preceq_{lex}}l_2') \quad \rightarrow$$

$$l_1{=}[x|l_1'] \wedge l_2{=}[y|l_2'] \wedge x{\leq}y,$$

and implement it as a CHR propagation rule, where the built-in inequality constraint appears in the body of a rule and is thus enforced when the rule is applied.

```
l4 @ [X|L1] lex [Y|L2] ==> X=<Y.
```

For example, to the query `[R|Rs] lex [T|Ts], R`$\neq$`T` only the propagation rule is applicable and adds `R=<T`. This results in `[R|Rs] lex [T|Ts], R<T` after simplification of the built-in constraints for inequality. Now rule `l2` is applicable, the `lex`-constraint is removed and the result is the remaining `R<T`.

However, the rules above are not sufficient, more propagations are possible. For example, consider `[R1,R2,R3] lex [T1,T2,T3], R2=T2, R3>T3`. The only way to satisfy this constraint is by asserting `R1<T1`, since the remaining elements cannot be ordered in the right way if `R1>=T1`. In order to perform such reasoning, we have to look forward, at the next level of recursion. If the recursive call fails, the base case for non-empty sequences must hold. If the recursive call proceeds to the next recursion, because the two variables are equal, we can ignore (i.e. remove) this pair of variables.

```
l5 @ [X,U|L1] lex [Y,V|L2] <=> U>V | X<Y.
l6 @ [X,U|L1] lex [Y,V|L2] <=> U=V | [X|L1] lex [Y|L2].
```

Admittedly, these last two rules require some insight.

Note that rules `l4` and `l5` are the only ones that directly impose a built-in constraint.

Our example, `[R1,R2,R3] lex [T1,T2,T3], R2=T2, R3>T3`, can now be handled. First, since `R2=T2`, rule `l6` is applicable, and its result is `R2=T2, R3>T3, [R1,R3] lex [T1,T3]`. Now rule `l5` is applied, and we arrive at `R2=T2, R3>T3, R1<T1` as desired.

There are still situations that are not covered by the current set of rules. Just replace `R2=T2` in the above example by `R2>=T2`. The same propagation should take place as before, but the two rules that we have added cannot be applied, their guards are too strict.

In these situations, the current pair of variables is followed by a sequence of variables which are pairwise in `>=` relation, which each could turn into a strict inequation later on. This can be covered by modifying rule `l6` into a propagation rule and weaken its guard:

```
l6'@ [X,U|L1] lex [Y,V|L2] ==> U>=V | [X|L1] lex [Y|L2].
```

The six rules (`l1` to `l5` and `l6'`) implement a complete constraint solver for the non-strict lexicographic order constraint for comparing sequences of the same, given length. The rules are obviously terminating. The recursions involve the `lex`-constraint only. Each recursive call proceeds with shorter lists (with one element less each). But the solver is not as efficient as it could be.

Improving Time Complexity In [11] it is shown that a cascade of propagation rule applications, and the subsequent simplification of the constraints produced, can lead to a quadratic time behavior with non-optimized CHR implementations. In order to regain linear time complexity, we try to replace the propagation rule `l6'` by an equally powerful simplification rule. Just changing `==>` into `<=>` results in a rule that is logically incorrect and also results in non-confluence.

A semantics-preserving translation of the propagation rule by adding the head to the body,

```
[X,U|L1] lex [Y,V|L2] <=> U>=V | [X,U|L1] lex [Y,V|L2], [X|L1] lex [Y|L2],
```

leads to obvious non-termination. But it gives us a clue in that it shows the problem of the repated occurences of the subsequences `L1` and `L2`. As it turns out (see Section on Correctness), removing `L1` and `L2` from the added `lex` constraint preserves correctness and improves time complexity to linear.

Our lexicographic constraint solver consists now of the following 6 rules.

```
l1 @ [] lex [] <=> true.
l2 @ [X|L1] lex [Y|L2] <=> X<Y | true.
l3 @ [X|L1] lex [Y|L2] <=> X=Y | L1 lex L2.
l4 @ [X|L1] lex [Y|L2] ==> X=<Y.
```

```
l5 @ [X,U|L1] lex [Y,V|L2] <=> U>V | X<Y.
l6''@[X,U|L1] lex [Y,V|L2] <=> U>=V, L1=[_|_] |
              [X,U] lex [Y,V], [X|L1] lex [Y|L2].
```

The additional condition `L1=[_|_]` in the guard of rule `l6''` avoids non-termination in case `L1=[]`.

Our algorithm is encoded by three pairs of rules, the first two corresponding to base cases of the recursion, then two rules performing the obvious recursive traversal of the sequences to be compared and finally two covering a not so obvious special case when the lexicographic constraint has a unique solution.

9 Example: The Union-Find Algorithm

The classical union-find (also: disjoint set union) algorithm was introduced by Tarjan in the seventies [26]. This essential algorithm efficiently solves the problem of maintaining a collection of disjoint sets under the operation of union [14]. It is the basis for many graph algorithms and for dealing with equality, e.g. in unification algorithms.

We have chosen union-find as an example, because it was recently shown how to implement it with optimal time complexity in CHR [24], something that is not known to be possible in other pure logic programming languages like Prolog. An analysis of the algorithm appears in [22], and a parallelization of the algorithm using confluence analysis is discussed in [12].

The union-find algorithm maintains disjoint sets under union. Each set is represented by a rooted tree, whose nodes are the elements of the set. The root is called the *representative* of the set. The representative may change when the tree is updated by a union operation. With the algorithm come three operations:

- `make(X)`: generate a new tree with the only node `X`, i.e. `X` is the root.
- `find(X)`: follow the path from the node `X` to the root of the tree by repeatedly going to the parent node of the current node until the root is reached. Return the root as representative.
- `union(X,Y)`: to join the two trees, find the representatives of `X` and `Y` (they are roots). Then `link` them by making one point to the other.

The following CHR program implements the operations and data structures of the basic union-find algorithm as CHR constraints [24].

```
make      @ make(A) <=> root(A).
union     @ union(A,B) <=> find(A,X), find(B,Y), link(X,Y).

findNode  @ A ~> B  \ find(A,X) <=> find(B,X).
findRoot  @ root(A) \ find(A,X) <=> X=A.

linkEq    @ link(A,A) <=> true.
link      @ link(A,B), root(A), root(B) <=> B ~> A, root(A).
```

The constraints `make/1`, `union/2`, `find/2` and `link/2` define the *operations*. The find operation is implemented as a relation `find/2` whose second argument returns the result. `link/2` is an auxiliary operation for performing union of two roots. The *tree (data) constraints* `root/1` and `~>/2` ("points to") represent the tree data structure.

The basic algorithm requires $\mathcal{O}(N)$ time per find (and union) in the worst case, where N is the number of elements (make operations). With two independent optimizations that keep the tree shallow and balanced, one can achieve logarithmic worst-case and quasi-constant (i.e. almost constant) amortized running time per operation.

The first optimization is *path compression* for find. It moves nodes closer to the root after a find. After `find(X)` returned the root of the tree, we make every node on the path from `X` to the root point directly to the root.

The second optimization is *union-by-rank*. It keeps the tree shallow by pointing the root of the smaller tree to the root of the larger tree. *Rank* refers to an upper bound of the tree depth (tree height). If the two trees have the same rank, either direction of pointing is chosen but the rank is increased by one. With this optimization, the height of the tree can be bound by $log(N)$. Thus the worst case time complexity for a single find or union operation is $\mathcal{O}(log(N))$.

The following CHR program implements these optimizations.

```
make      @ make(A) <=> root(A,0).
union     @ union(A,B) <=> find(A,X), find(B,Y), link(X,Y).

findNode  @ A ~> B, find(A,X) <=> find(B,X), A ~> X.
findRoot  @ root(A,_) \ find(A,X) <=> X=A.

linkEq    @ link(A,A) <=> true.
linkLeft  @ link(A,B), root(A,N), root(B,M) <=> N>=M |
                B ~> A, N1 is max(N,M+1), root(A,N1).
linkRight @ link(B,A), root(A,N), root(B,M) <=> N>=M |
                B ~> A, N1 is max(N,M+1), root(A,N1).
```

When compared to the basic version of the algorithm implementation, we see that `root` has been extended with a second argument that holds the rank of the root node.

The rule `findNode` has been extended for immediate path compression: the logical variable `X` serves as a place holder for the result of the find operation. The `link` rule has been split into two rules `linkLeft` and `linkRight` to reflect the optimization of union-by-rank: The smaller ranked tree is added to the larger ranked tree without changing its rank. When the ranks are the same, either tree is chosen (both rules are applicable) and the rank is incremented.

10 Summary and Outlook

You will hear it at the tutorial. Instead here are some late breaking news: The paper [17] introduces CHR machines, analogous to RAM and Turing machines.

It shows that these machines can simulate each other in polynomial time, thus establishing that CHR is turing-complete and, more importantly, that every algorithm can be implemented in CHR with best known time and space complexity.

References

1. S. Abdennadher. Operational Semantics and Confluence of Constraint Propagation Rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP97*, LNCS 1330. Springer, 1997.
2. S. Abdennadher and T. Frühwirth. On completion of constraint handling rules. In *4th International Conference on Principles and Practice of Constraint Programming, CP98*, LNCS 1520. Springer, 1998.
3. S. Abdennadher and T. Frühwirth. Operational equivalence of constraint handling rules. In *Fifth International Conference on Principles and Practice of Constraint Programming, CP99*, LNCS 1713. Springer, 1999.
4. S. Abdennadher, T. Frühwirth, and C. H. (Eds.). *Special Issue on Constraint Handling Rules, Journal of Theory and Practice of Logic Programming (TPLP).* Cambridge University Press, to appear 2005.
5. S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and Semantics of Constraint Simplification Rules. *Constraints Journal*, 4(2), 1999.
6. H. Betz and T. Frühwirth. A Linear-Logic Semantics for Constraint Handling Rules. In *International Conference on Principles and Practice of Constraint Programming (CP05)*, LNCS. Springer, 2005.
7. G. J. Duck, P. J. Stuckey, M. G. de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, 2004.
8. T. Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, pages 95–138, October 1998.
9. T. Frühwirth. Proving termination of constraint solver programs. In E. M. K.R. Apt, A.C. Kakas and F. Rossi, editors, *New Trends in Constraints*, LNAI 1865. Springer, 2000.
10. T. Frühwirth. As Time Goes By: Automatic Complexity Analysis of Simplification Rules. In *8th International Conference on Principles of Knowledge Representation and Reasoning*, Toulouse, France, 2002.
11. T. Frühwirth. Logical Rules for a Lexicographic Order Constraint Solver. In *Second Workshop on Constraint Handling Rules, at ICLP05*, Sitges, Spain, October 2005.
12. T. Frühwirth. Parallelizing Union-Find in Constraint Handling Rules Using Confluence. In *International Conference on Logic Programming (ICLP05)*, LNCS. Springer, 2005.
13. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming.* Springer, 2003.
14. Z. Galil and G. F. Italiano. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Comp. Surveys*, 23(3):319ff, 1991.
15. J.-Y. Girard. Linear logic: Its syntax and semantics. *Theoretical Computer Science*, 50:1–102, 1987.

16. J.-L. J. Imbert. Linear constraint solving in clp-languages. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910, Berlin, Heidelberg, New York, 1995. Springer.
17. B. D. Jon Sneyers, Tom Schrijvers. The Computational Power and Complexity of Constraint Handling Rules. In *Second Workshop on Constraint Handling Rules, at ICLP05*, Sitges, Spain, October 2005.
18. S. Menju, K. Sakai, Y. Sato, and A. Aiba. A study on boolean constraint solvers. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 253–268. MIT Press, Cambridge, Mass., 1993.
19. B. D. Peter Van Weert, Tom Schrijvers. he K.U.Leuven JCHR System. In *Second Workshop on Constraint Handling Rules, at ICLP05*, Sitges, Spain, October 2005.
20. M. S. S. Abdennadher, E. Krämer and M. Schmauss. Jack: A java constraint kit. In *Electronic Notes in Theoretical Computer Science*, volume 64, 2000.
21. V. A. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–352, New York, NY, USA, 1991. ACM Press.
22. T. Schrijvers and T. Frühwirth. Analysing the CHR Implementation of Union-Find. In *19th Workshop on (Constraint) Logic Programming (W(C)LP 2005).* Ulmer Informatik-Berichte 2005-01, University of Ulm, Germany, February 2005.
23. T. Schrijvers and T. Frühwirth. CHR Website, `www.cs.kuleuven.ac.be/~dtai/projects/CHR/`, May 2005.
24. T. Schrijvers and T. Frühwirth. Optimal Union-Find in Constraint Handling Rules, Programming Pearl. *Journal of Theory and Practice of Logic Programming (TPLP)*, to appear.
25. T. Schrijvers, P. J. Stuckey, and G. J. Duck. Abstract interpretation for constraint handling rules. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 218–229, New York, NY, USA, 2005. ACM Press.
26. R. E. Tarjan and J. van Leeuwen. Worst-case Analysis of Set Union Algorithms. *J. ACM*, 31(2):245–281, 1984.

Appendix: Additional Selected Recent Constraint Handling Rules Papers by Topic

Since this paper does not discuss implementation and application of CHR, here is a list of publications in those areas. Papers where chosen in June 2005 based on contents, popularity, publication type, recency, presentation. Most paper are available with links at the CHR webpages [23].

Implementation/Compilation/Transformation/Extension of CHR

COMPILATION/IMPLEMENTATION:

• T. Schrijvers, B. Demoen, G. Duck, P. Stuckey, and T. Frühwirth, Automatic implication checking for CHR constraints, Proc. of 6th Intl. Workshop on Rule-Based Programming, Nara, Japan, April, 2005.
∗ J. Sneyers, T. Schrijvers and B. Demoen, Guard Simplification in CHR programs, 9th Workshop on (Constraint) Logic Programming, Ulm, February, 2005.

∗ Tom Schrijvers and Bart Demoen, The K.U.Leuven CHR system: implementation and application, 1st WS on CHR, University of Ulm, Ulmer Informatik-Berichte Nr. 2004-01, May 2004.
• C. Holzbaur, P.J. Stuckey, M. Garcia de la Banda, and D. Jeffery, Optimizing compilation of constraint handling rules, Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules, to appear 2005.
∗ Gregory J. Duck, Peter .J. Stuckey, Maria Garcia de la Banda, and Christian Holzbaur, Extending arbitrary solvers with constraint handling rules, (D. Miller, Ed.), Proc. of the Fifth ACM SIGPLAN Intl. Conf. on Principles and Practice of Declarative Programming, ACM Press, 2003.
• C. Holzbaur and T. Frühwirth, A Prolog Constraint Handling Rules Compiler and Runtime System, Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules (C. Holzbaur and T. Frühwirth, Eds.), Taylor & Francis, Vol 14(4), April 2000.
• S. Abdennadher, E. Krämer, M. Saft and M. Schmauss, JACK: A Java Constraint Kit, Electronic Notes in Theoretical Computer Science Volume 64, 2000.

RULE GENERATION:

• Slim Abdennadher, Christophe Rigotti, Automatic Generation of CHR Constraint Solvers, Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules, to appear 2005. Many more publications, e.g. CP 2000.
• Sebastian Brand, Krzysztof Apt, Schedulers and Redundancy for a Class of Constraint Propagation Rules, Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules, to appear 2005.
• KR Apt, E Monfroy, Constraint programming viewed as rule-based programming, TPLP Journal, 2001.
∗ KR Apt, E Monfroy, Automatic Generation of Constraint Propagation Algorithms for Small Finite Domains, CP, 1999.

PARTIAL EVALUATION/PROGRAM TRANSFORMATION:

• Thom Frühwirth, Specialization of Concurrent Guarded Multi-Set Transformation Rules, LOPSTR 2004 Intl. Symposium on Logic-based Program Synthesis and Transformation 2004, Verona, Italy, August 2004.
∗ T. Frühwirth and C. Holzbaur, Source-to-Source Transformation for a Class of Expressive Rules, Joint Conf. on Declarative Programming APPIA-GULP-PRODE 2003 (AGP 2003), Reggio Calabria, Italy, September 2003.

SOLVER COOPERATION/INTEGRATION:

• Eric Monfroy, Carlos Castro, Basic Components for Constraint Solver Cooperations, ACM SAC'03, ACM Press, Florida, USA, 2003.
• S. Abdennadher and T. Frühwirth, Integration and Optimization of Rule-Based Constraint Solvers, Logic Based Program Synthesis and Transformation - LOPSTR 2003, Revised Selected Papers, (M. Bruynooghe, ed.), LNCS 3018, Springer Verlag, 2004.

EXTENSIONS OF CHR:

• Constraint Handling Rules and Tabled Execution, Tom Schrijvers, David S. Warren, ICLP'04 20th Intl. Conf. on Logic Programming, September 6-10, 2004, Saint-Malo, France. Best Paper Award.

• T. Frühwirth, A. Di Pierro and H. Wiklicky, Probabilistic Constraint Handling Rules, 11th Intl. Workshop on Functional and (Constraint) Logic Programming (WFLP 2002), Selected Papers, Guest Eds.: Marco Comini and Moreno Falaschi, Vol. 76 of Electronic Notes in Theoretical Computer Science (ENTCS), 2002.
• Armin Wolf, Intelligent Search Strategies Based on Adaptive Constraint Handling Rules, Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules, to appear 2005.
∗ Armin Wolf, DJCHR - A Java-Based System for Dynamic Constraint Handling - Theory, Architectur and Application, Invited Talk, First CHR Workshop, University of Ulm, May 2004.
∗ Armin Wolf, Adaptive Constraint Handling with CHR in Java, CP, LNCS 2239, 2001.
∗ Armin Wolf, Thomas Gruenhagen, and Ulrich Geske, On the incremental adaptation of CHR derivations, Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules (C. Holzbaur and T. Frühwirth, Eds.), Taylor & Francis, Vol 14(4), April 2000.
• S Abdennadher, H Christiansen, An Experimental CLP Platform for Integrity Constraints and Abduction, FQAS 2000.
∗ S Abdennadher, H Schuetz, CHRv: A Flexible Query Language, FQAS 1998.
• E. Coquery and F. Fages, Un systeme de types pour CHR, (C. Solnon, Ed.), Actes des Journes Francophones de la Programmation par Contraintes JFPC'2005, AFPC, June 2005.
∗ Francois Fages and Emmanuel Coquery, Typing Constraint Logic Programs, Journal of Theory and Practice of Logic Programming TPLP 1(6), pp. 751-777, November 2001.

Popular Application Areas of CHR

AGENTS/ACTIONS:

• Michael Thielscher, FLUX: A Logic Programming Method for Reasoning Agents, Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules, to appear 2005.
• Christian Seitz, Bernhard Bauer, Michael Berger, Multi Agent Systems Using Constrains Handling Rules, IC-AI 2002, Las Vegas, Nevada, USA, 2002.
• Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni, Compliance Verification of Agent Interaction: a Logic-Based Tool, (R. Trappl, Ed.), Proc. of the 17th European Meeting on Cybernetics and Systems Research (EMCSR'2004), Vol. II, Symposium "From Agent Theory to Agent Implementation" (AT2AI-4), pp. 570-575, Vienna, Austria, April 2004.
∗ M Alberti, M Gavanelli, E Lamma, P Mello et. al., Specification and verification of agent interactions using social integrity constraints, Electronic Notes in Theoretical Computer Science, 2003.
∗ M Gavanelli, E Lamma, P Mello, M Milano, P Torroni, Interpreting abduction in CLP, AGP, Reggio Calabria, Italy 2003.

GRAMMARS/PARSING/COMPTUATIONAL LINGUISTICS:

• Henning Christiansen, CHR Grammars, Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules, to appear 2005.

∗ Dulce Aguilar-Solis, Veronica Dahl, Coordination Revisited: A Constraint Handling Rule Approach, Springer LNCS 3315, 2004.
• Topological Parsing, Gerald Penn and Mohammad Haji-Abdolhosseini, Proc. of the 10th Conf. of the European Chapter of the Association for Computational Linguistics (EACL-03), Budapest, Hungary, April 2003.

TYPES:

• PJ Stuckey, M Sulzmann, A theory of overloading, To appear in ACM Transactions on Programming Languages and Systems.
∗ Gregory J. Duck, Simon Peyton Jones, Peter J. Stuckey, and Martin Sulzmann, Sound and Decidable Type Inference for Functional Dependencies, European Symposium on Programming 2004 (ESOP'04), (D. Schmidt, ed.), LNCS 2968, Springer Verlag, 2004.
∗ Martin Sulzmann et. al., The Chameleon System, 1st WS on CHR, University of Ulm, Ulmer Informatik-Berichte Nr. 2004-01, May 2004.
∗ K Glynn, PJ Stuckey, M Sulzmann, Type Classes and Constraint Handling Rules,...
∗ Martin Sulzmann and Jeremy Wazny, Enforcing Security Policies using Overloading Resolution, The University of Melbourne, Department of Computer Science, TR2001/32, October 2001.
• Emmanuel Coquery and Franois Fages, Subtyping Constraints in Quasi-lattices, (P. K. Pandya and J. Radhakrishnan, Eds.), Foundations of Software Technology and Theoretical Computer Science, FSTTCS'03, volume 2914 of LNCS, pages 136-148, Springer-Verlag, 2003.

SCHEDULING/SPATIAL AND TEMPORAL REASONING:

• S. Abdennadher, M. Marte, University Course Timetabling Using Constraint Handling Rules, Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules (C. Holzbaur and T. Frühwirth, Eds.), Taylor & Francis, Vol 14(4), April 2000.
∗ S. Abdennadher, M. Saft, S. Will, Classroom Assignment using Constraint Logic Programming, PACLP 2000.
• MT Escrig, F Toledo, Autonomous robot navigation using human spatial concepts, Intl. Journal of Intelligent Systems, 2000.
∗ Escrig, M. T. and Toledo, F., A Framework Based on CLP Extended with CHRs for Reasoning with Qualitative Orientation and Positional Information, Journal of Visual Languages and Computing, 9, 1998, p. 81-101.
∗ Escrig, M. T. and Toledo, F.: Qualitative Spatial Reasoning : Theory and Practice - Application to Robot Navigation. Frontiers in AI and Applications, Volume 47, IOS Press, Amsterdam, 1998.
• Bernd Meyer, A constraint-based framework for diagrammatic reasoning, Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules (C. Holzbaur and T. Frühwirth, Eds.), Taylor & Francis, Vol 14(4), April 2000.
• Thom Frühwirth, Temporal Reasoning with Constraint Handling Rules, ECRC TR-94-5, Munich, 1994.

SEMANTIC WEB:

• Hongwei Zhu, Stuart E. Madnick and Michael D. Siegel, Reasoning About Temporal Context Using Ontology and Abductive Constraint Logic Programming, Principles and Practice of Semantic Web Reasoning: Second Intl. Workshop, PPSWR 2004, St. Malo,

France, September, Springer LNCS 3208, 2004.
∗ Hongwei Zhu, Stuart E. Madnick, Michael Siegel, Effective Data Integration in the Presence of Temporal Semantic Conflicts, TIME 2004:109-114, 2004.
∗ A Firat, S Madnick, B Grosof, Knowledge Integration to Overcome Ontological Heterogeneity: Challenges from Financial Information..., Proc. of ICIS, 2002.
∗ Stephane Bressan, Cheng Hian Goh, Natalia Levina, Stuart E. Madnick, Ahmed Shah, Michael Siegel, Context Knowledge Representation and Reasoning in the Context Interchange System, Appl. Intell. 13(2):165-180, 2000.
∗ Cheng Hian Goh, Stephane Bressan, Stuart E. Madnick, Michael Siegel, Context Interchange: New Features and Formalisms for the Intelligent Integration of Information, ACM Trans. Inf. Syst. 17(3): 27-293, 1999.
• Liviu Badea, Doina Tilivea and Anca Hotaran, Semantic Web Reasoning for Ontology-Based Integration of Resources, Principles and Practice of Semantic Web Reasoning: Second Intl. Workshop, PPSWR 2004, St. Malo, France, September, Springer LNCS 3208, 2004.
• J van Ossenbruggen, J Geurts, F Cornelissen, L Hardman, L Rutledge, Towards Second and Third Generation Web-Based Multimedia, The Tenth Intl. World Wide Web Conf., ACM Press, 2001. 60 citations.
∗ Joost Geurts, Jacco van Ossenbruggen, Lynda Hardman, Application-Specific Constraints for Multimedia Presentation Generation, Proc. of the Intl. Conf. on Multimedia Modeling 2001 (MMM01), p. 247-266, CWI, Amsterdam, The Netherlands, November 2001.
• 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, April 1995.

SOFTWARE ENGINEERING/TESTING:

• Model Based Testing for Real: The Inhouse Card Case Study, A. Pretschner, O. Slotosch, E. Aiglstorfer and S. Kriebel, Intl. Journal on Software Tools for Technology Transfer (STTT), Volume 5, Numbers 2-3, Springer Verlag, March 2004.
∗ A Pretschner, Classical search strategies for test case generation with Constraint Logic Programming, Proc. Formal Approaches to Testing of Software, 2001.
∗ H Lotzbeyer, A Pretschner, Testing Concurrent Reactive Systems with Constraint Logic Programming, Proc. 2nd workshop on Rule-Based Constraint Reasoning..., 2000.
∗ H Lotzbeyer, A Pretschner, AutoFocus on Constraint Logic Programming, Proc. (Constraint) Logic Programming and Software Engineering, London, 2000.