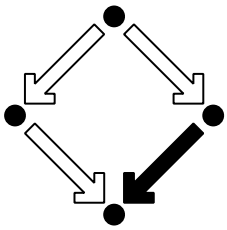
# CHR 2008

## The 5th Workshop on Constraint Handling Rules

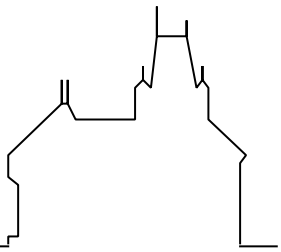
# Proceedings

**Editors: Tom Schrijvers, Frank Raiser, Thom Frühwirth**

**July 14, 2008**
**Castle of Hagenberg, Austria**



**RISC-Linz**
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe



# CHR 2008

## The 5th Workshop on Constraint Handling Rules

Tom SCHRIJVERS, Frank RAISER,
Thom FRÜHWIRTH (editors)

Castle of Hagenberg, Austria
July 14, 2008

RISC-Linz Report Series No. 08-10



Supported by: Linzer Hochschulfonds, Upper Austrian Government, Austrian Federal Ministry of Science and Research (BMWF), Johann Radon Institute for Computational and Applied Mathematics of the Austrian Academy of Sciences (RICAM)

# Preface

This book contains the proceedings of CHR 2008, the fifth workshop on Constraint Handling Rules, held at the occasion of RTA 2008 in Hagenberg (Austria) on July 14, 2008.

The Constraint Handling Rules (CHR) language has become a major declarative specification and implementation language for constraint reasoning algorithms and applications. Algorithms are often specified using inference rules, rewrite rules, proof rules, or logical axioms that can be directly written in CHR. Based on first-order predicate logic, this clean semantics of CHR facilitates non-trivial program analysis and transformation.

Previous Workshops on Constraint Handling Rules were organized in May 2004 in Ulm (Germany), in October 2005 in Sitges (Spain) at ICLP, in July 2006 in Venice (Italy) at ICALP, and in September 2007 in Porto (Portgual) at ICLP. It means to bring together in an informal setting, people involved in research on all matters involving CHR, in order to promote the exchange of ideas and feedback on recent developments.

Twelve papers were submitted to the workshop, all of which were carefully reviewed by three reviewers. Ten papers and one short paper were accepted to the workshop.

We are grateful to all the authors of the submitted papers, the program committee members, and the referees for their time and efforts spent in the reviewing process, and the RTA 2008 organizers for hosting our workshop.

July 2008

Tom Schrijvers
Frank Raiser
Thom Frühwirth

# Organization

## Workshop Coordinators:

- Tom Schrijvers (Belgium)
- Frank Raiser (Germany)
- Thom Frühwirth (Germany)

## Program Committee:

- François Fages, INRIA Rocquencourt
- Peter J. Stuckey, NICTA Victoria Laboratory
- Jacques Robin, Universidade Federal de Pernambuco
- Martin Sulzmann, National University of Singapore
- Maurizio Gabbrielli, Universita di Bologna
- Slim Abdennadher, German University in Cairo
- Thom Frühwirth, Universität Ulm
- Tom Schrijvers, Katholieke Universiteit Leuven
- Armin Wolf, Fraunhofer FIRST, Berlin
- Verónica Dahl, Simon Fraser University in Vancouver
- Beata Sarna-Starosta, Michigan State University
- Evelina Lamma, Università di Ferrara

## Referees:

| | | |
|---|---|---|
| Marcus Aurelio | Leslie De Koninck | Jon Sneyers |
| Thierry Martinez | Gregory J. Duck | Frank Raiser |
| Peter Van Weert | Luc Maranget | Marco Alberti |
| Marco Gavanelli | Hariolf Betz | Ingi Sobhi |
| Cinzia Di Giusto | | |

# Table of Contents

# The CHR-Celf Connection

Anders Schack-Nielsen and Carsten Schürmann

IT University of Copenhagen
Denmark

**Abstract.** Celf is a meta-language for specifying and implementing deductive and concurrent systems from areas, such as programming language theory, process algebras, and logics. It is based on the concurrent logical framework CLF [CPWW02a]. The Constraint Handling Rules (CHR) language [Frü98] is a major specification and implementation language for constraint-based algorithms.
In this invited talk, we give a tutorial-style introduction to Celf for the CHR programmer where we highlight some of Celf's features including the support of higher-order encodings, first-class execution traces, and a logically inspired proof search semantics. Furthermore we show where the semantics of the two languages coincide.

The Celf system is a tool for experimenting with deductive and concurrent systems prevalent in programming language theory, process algebras, and logics. It supports the specification of object language syntax and semantics through a combination of deductive methods and resource-aware concurrent multiset transition systems. Furthermore it supports the experimentation with those specifications through concurrent logic programming based on multiset rewriting.

Many case studies have been conducted in Celf including all of the motivating examples that were described in the original CLF technical report [CPWW02b]. In particular, Celf has been successfully employed for experimenting with concurrent ML, its type system, and a destination passing style operational semantics that includes besides the pure core a clean encoding of Haskell-style suspensions with memoizations, futures, mutable references, and concurrency omitting negative acknowledgments. Other examples include various encodings of the $\pi$-calculus, security protocols, petri-nets, etc.

CLF is a conservative extension over LF, which implies that Celf's functionality is compatible with that of Twelf [PS99]. With a few syntactic modifications Twelf signatures can be read, type checked, and queries can be executed. Celf does not yet provide any of the meta-theoretic capabilities that sets Twelf apart from its predecessor Elf, such as mode checking, termination checking, coverage checking, and the like, which we leave to future work. In this presentation we concentrate on the two main features of Celf.

*Specification.* CLF was designed with the objective in mind to simplify the specification of object languages by internalizing common concepts used for specification. Celf supports dependent types for the encoding of judgments as types, e.g. typing relations between terms and types, operational relations between

terms and values, open and closed terms, derivability, and logical truth. It also supports the method of *higher-order abstract syntax*, which relieves the user of having to specify substitutions and substitution application. In CLF, every term is equivalent to a unique inductively defined $\beta$-normal $\eta$-long form modulo $\alpha$-renaming and let-floating providing an induction principle to reason about the adequacy of the encoding. In addition, CLF provides linear types and concurrency encapsulating monadic types in support of the specification of resource aware and concurrent systems. Examples include operational semantics for languages with effects, transition systems, and protocol stacks.

*Experimentation.* Celf provides a logic programming interpreter that implements a proof search algorithm for derivations in CLF type theory in analogy to how Elf implements a logical programming interpreter based on uniform proof search. Celf's interpreter is inspired (with few modifications) by Lollimon [LPPW05], an extension of Lolli, the linear sibling of $\lambda$-Prolog. The interpreter implements backward-chaining search within the intuitionistic and linear fragment of CLF and switches to forward-chaining multiset rewriting search upon entering the monad. Celf programs may jump in and out of the concurrency monad and can therefore take advantage of both modes of operation. In addition, the operational semantics of Celf is conservative over the operational semantics of Elf, which means that any Twelf query can also be executed in Celf leading to the same result.

Celf is written in Standard ML and compiles with SML/NJ, MLton and MLKit. The source code and a collection of examples are freely available from http://www.twelf.org/∼celf.

## References

[CPWW02a] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University. Department of Computer Science, 2002.

[CPWW02b] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Carnegie Mellon University. Department of Computer Science, 2002.

[Frü98] Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998.

[LPPW05] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In Pedro Barahona and Amy P. Felty, editors, *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 35–46, Lisbon, Portugal, 2005.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

# Transformation-based Indexing Techniques for Constraint Handling Rules

Beata Sarna-Starosta
and Tom Schrijvers*

[1] Department of Computer Science and Engineering, Michigan State University, USA
LogicBlox Inc., Atlanta, Georgia, USA
bss@cse.msu.edu
[2] Department of Computer Science, K.U.Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

**Abstract.** Multi-headed rules are essential for the expressiveness of Constraint Handling Rules (CHR), but incur considerable performance overhead. Current indexing techniques are often unable to address this problem—they require matchings to have particular form, or offer good run-time complexity rather than good absolute figures.
We introduce two lightweight program transformations, based on term flattening, which improve the effectiveness of existing CHR indexing techniques, in terms of both complexity and constant factors. We also describe a set of complementary post-processing program transformations, which considerably reduce the flattening overhead.
We compare our techniques with the current state of the art in CHR compilation, and measure their efficacy in K.U.Leuven CHR and CHRd.

## 1 Introduction

Constraint Handling Rules (CHR) [1] is a high-level rule-based declarative programming language, usually embedded in a host language such as Prolog or Haskell. Typical applications of CHR include scheduling [2] and type checking [3]. CHR features *multi-headed rules*, i.e., rules with multiple predicates on the left-hand side (the *head*), which sets it apart from conventional declarative languages, e.g., Prolog or Haskell, where a rule's head admits only one predicate or function.

Multi-headed rules afford much of CHR's expressive power by allowing to easily combine information from distinct constraints via matching. However, as the matching procedure significantly affects the complexity of rule evaluation, this source of expressiveness often leads to performance bottlenecks. This effect is borne out by the approximative complexity formula of [4], where the multiplicity of rule's head appears in the exponent.

Aware of this problem, CHR developers have built data structures to support efficient indexing on variables (attributed variables [5]) and ground data

* Post-Doctoral Researcher of the Fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen)

(search trees [6]). With [7] came the realization that $\mathcal{O}(1)$ indexing is essential to implement CHR algorithms with optimal complexity, which led to the use of hash tables for indexing ground data, and the general result that the complexity of CHR systems equals that of RAM machines [8]. CHRd [9] has slimmed the original indexing techniques based on attributed variables for faster evaluation of the class of direct-indexed CHR and use in a tabulated environment.

In this paper we advance the research on CHR indexing with the following contributions:

- two independent program transformations that improve the indexing behavior of CHR in the presence of function symbols
- a sequence of program post-processing steps that eliminate the overhead incurred by the indexing transformations
- the experimental measurements that clearly demonstrate the potential for time complexity improvements and the practical usefulness of our approach in K.U.Leuven CHR and CHRd
- the implementation of the presented transformations[1]

Our presentation begins with an overview of the problem of indexing with partial structures (Section 2). We then introduce our two indexing techniques (Sections 3 and 4), and describe applicable post-processing steps (Section 5). Next, we present the experimental evaluation of all proposed transformations (Section 6), relate our approach to other work (Section 7), and conclude (Section 8).

## 2 Problem Overview

*The Problem* CHR systems build indexes on the constraint store to speed up matching multi-headed rules. Consider the rule:

```
a(X), b(X,Y) ==> write(Y).
```

Given `X`, an index returns all stored constraints `b(X,Y)`. Thus, for a new `a(X)` we can quickly find all matchings of the form `b(X,Y)` that make the rule fire. Now consider the variant of the previous rule:

```
a(X), b(f(_,_),Y) ==> write(Y).
```

Here, for efficient matching, we need an index that returns all instances of `b/2` in which the first argument has top-level function symbol `f/1`. Currently, CHR systems do not generate indexes that involve partial structure of the constraints. Instead, they enumerate all `b(A,B)` in the constraint store and, for each `A`, test whether its top-level function symbol is `f/1`. When only a small fraction of all

[1] available at `http://www.cs.kuleuven.be/~toms/CHR/Indexing/`

As have this form, there are many failing tests. The problem becomes even more apparent if partial structures are parameterized, as in the rule:

```
a(X), b(f(X,_),Y) ==> write(Y).
```
(2.1)

Existing CHR systems cannot exploit the variable `X` to find all stored constraints matching `b(f(X,_),Y)` more quickly: as before, finding the matchings for an active constraint `a(X)` requires that all stored `b/2` constraints are enumerated.

*The Solution* We propose two techniques—both based on *term flattening*—for building indexes on partial structures. The first, *generic flattening* (Section 3), transforms rule (2.1) into:

```
a(X), b'(f,X,_,Y) ==> write(Y),
```

and the second, *constraint symbol specialization* (Section 4), into:

```
a(X), b_f(X,_,Y) ==> write(Y).
```

As source-to-source transformations, both proposed techniques are portable to many CHR systems. Moreover, since they both reuse available indexing data structures, further optimizations of such data structures also improve the indexing performance of our techniques. As we prove in [10], both proposed techniques preserve the theoretical [1] and refined [11] semantics of CHR, as well as the set-based semantics of CHRd [9].

*Preliminaries* We restrict our presentation to CHR programs where each rule head contains at most one occurence of a functional term, at a fixed argument position of some constraint $c/n$. We consider the $i$th argument of $c/n$, and a given set $F$ of function symbols $f_j/a_j$ that appear in the rule heads at the $i$th position of $c/n$. We define the *maximal arity* of $F$ as $a_{max} = \max_{f_j/a_j \in F}(a_j)$.

We assume that, at run time, all instances of $c/n$ have the top-level function symbol in their $i$th argument instantiated, but not necessarily to a symbol in $F$. This assumption can be satisfied by groundness analysis [12] or programmer-supplied mode annotations.

*Example 1.* The first argument of the constraint $c/1$ from the CHR program in Table 1 takes on function symbols given by the set $F = \{f/2, g/1\}$. The maximal arity of $F$, $a_{max}$, is 2.

## 3 Generic Flattening

Our first flattening approach augments the arity of each constraint symbol $c$, which appears in the heads of the program rules with function-term arguments, to accommodate new arguments of $c$ representing these function terms.

```
:- chr_constraint c/1.

r1 @ c(X) \ c(X) <=> true.
r2 @ c(f(X,Y))   ==> c(X), c(Y).
r3 @ c(g(X))     ==> c(X).
r4 @ c(X)        <=> write(X).
```

**Table 1.** A CHR program with constraints on function symbols

**Definition 1 (Flattening and Unflattening Functions).** *The* flattening function $\phi$ *with respect to the set of function symbols* $F$*, maps a term* $T$ *onto a sequence of terms:*

$$\phi(T) = \begin{cases} f_j, \bar{t}, \underbrace{e, \ldots, e}_{d_j} & \text{if } T = f_j(\bar{t}) \text{ s.t. } f_j/a_j \in F \text{ and } |\bar{t}| = a_j \\ T, \underbrace{e, \ldots, e}_{a_{max}} & \text{otherwise} \end{cases}$$

*where* $d_j = a_{max} - a_j$ *and* $e$ *is an arbitrary constant.*

*The* unflattening function $\psi = \phi^{-1}$ *maps a sequence of terms onto a term:*

$$\psi(T) = \begin{cases} f_j(\bar{t}) & \text{if } T = f_j, \bar{t}, \bar{e} \text{ s.t. } f_j/a_j \in F \text{ and } |\bar{t}| = a_j \text{ and } |\bar{e}| = a_{max} - a_j \\ t & \text{if } T = t, \bar{e} \text{ s.t. } |\bar{e}| = a_{max} \end{cases}$$

*Example 2.* The flattening and unflattening functions for the CHR program in Table 1, with $F = \{f/2, g/1\}$, are defined as:

$$\phi(T) = \begin{cases} f, X, Y & \texttt{if } T = f(X,Y) \\ g, Z, e & \texttt{if } T = g(Z) \\ T, e, e & \texttt{otherwise} \end{cases}$$

$$\psi(A,B,C) = \begin{cases} f(X,Y) & \texttt{if } A,B,C = f,X,Y \\ g(Z) & \texttt{if } A,B,C = g,Z,e \\ T & \texttt{if } A,B,C = T,e,e \end{cases}$$

The original and flattened instances of the constraints are related by the flattening rules:

**Definition 2 (Flattening Rule).** *The* flattening rule $\Phi$ *replaces a given constraint* $c/n$ *with its flat form:*

$$\Phi \ \texttt{@} \ c(\overline{t_{1,\ldots,i-1}}, t_i, \overline{t_{i+1,\ldots,n}}) \ \texttt{<=>} \ c'(\overline{t_{1,\ldots,i-1}}, \phi(t_i), \overline{t_{i+1,\ldots,n}}) \tag{3.2}$$

*Example 3.* The flattening rule for constraint `c/1` of the CHR program in Table 1 is listed in line 3 of Table 2.

```
:- chr_constraint c/1, c'/3.                                    1
                                                                2
 Φ  @ c(T) <=> c'(φ(T)).                                        3
                                                                4
r1' @ c'(A,B,C) \ c'(A,B,C) <=> true.                           5
r2' @ c'(f,X,Y) ==> c(X), c(Y).                                 6
r3' @ c'(g,X,e) ==> c(X).                                       7
r4' @ c'(A,B,C) <=> T = ψ(A,B,C) | write(T).                    8
```

**Table 2.** CHR program from Table 1 after generic flattening

**Definition 3 (Flattened Rule).** *The flattening $\phi$ of a CHR rule with respect to the ith argument of a constraint $c/n$ for a set of function symbols $F$ is defined as:*

$$\phi(H \texttt{ ?=> } G \mid B) \;=\; H' \texttt{ ?=> } G', G \mid B$$

*where*

- *$H'$ differs from $H$ in that any constraint $c(t_1, \ldots, t_i, \ldots, t_n)$ is replaced by $c'(t_1, \ldots, t'_i, \ldots, t_n)$ where $t'_i = \phi(t_i)$ if $t_i = f_j(t'_1, \ldots, t'_{a_j})$ s.t. $f_j/a_j \in F$, or $t'_i = \bar{t}$, where $\bar{t}$ are fresh variables, otherwise*
- *the new guard $G'$ relates the flattened arguments back to the original ones, and contains one $t_i$ `=` $\psi(t'_i)$ for each flattened argument.*

*Example 4.* Lines 5-8 of Table 2 list flattened rules of the CHR program in Table 1, partially post-processed (Section 5) for readability.

**Definition 4 (Flattened Program).** *The flattening $\phi(\mathcal{P})$ of a CHR program $\mathcal{P}$ given by the set of rules $\overline{R}$, with respect to the ith argument of a constraint $c/n$, for a set of function symbols $F$, is defined as the flattening of each rule in $\overline{R}$, the flattening and unflattening functions, and the encoding of $\Phi$:*

$$\phi(P) = \phi(\overline{R}) \cup \phi \cup \psi \cup \Phi$$

## 4 Constraint Symbol Specialization

Our second flattening technique differs from the first one in that it uses a different constraint symbol for each function symbol. As a consequence, it defines one flattening function and multiple unflattening functions:

**Definition 5 (Flattening and Unflattening Functions).** *The* flattening function *$\phi$ with respect to the set of function symbols $F$, maps a term $T$ onto a sequence of terms:*

$$\phi(T) = \begin{cases} \bar{t} & \text{if } T = f_j(\bar{t}) \text{ s.t. } f_j/a_j \in F \text{ and } |\bar{t}| = a_j \\ T & \text{otherwise} \end{cases}$$

*The* unflattening function *for a function symbol $f_j/a_j$, $\psi_{f_j}$, maps a sequence of terms onto a term: $\psi_{f_j}(t_1, \ldots, t_{a_j}) = f_j(t_1, \ldots, t_{a_j})$. The* default unflattening function *is the identity function': $\psi'(t) = t$.*

*Example 5.* The flattening and unflattening functions for the CHR program in Table 1, with $F = \{f/2, g/1\}$, are defined as:

$$\phi(T) = \begin{cases} X, Y & \texttt{if } T = f(X,Y) \\ Z & \texttt{if } T = g(Z) \\ T & \texttt{otherwise} \end{cases}$$

$$\begin{aligned} \psi_f(X,Y) &= f(X,Y) \\ \psi_g(Z) &= g(Z) \\ \psi'(T) &= T \end{aligned}$$

Note that the flattening function is the *left inverse* of each unflattening function:

$$\phi \circ \psi_{f_j}(\bar{t}) = \bar{t} \tag{4.3}$$
$$\phi \circ \psi'(t) = t \tag{4.4}$$

and it is the *right inverse* for terms with appropriate function symbols:

$$\psi_{f_j} \circ \phi(f_j(\bar{t})) = f_j(\bar{t}) \qquad \text{where } f_j \in F \tag{4.5}$$
$$\psi' \circ \phi(f(\bar{t})) = f(\bar{t}) \qquad \text{where } f \notin F \tag{4.6}$$

Hence, unlike in the previous section, the flattening function is (in general) not injective: the original term $t$ cannot be reconstructed from its flattened form $\bar{t}$ alone. Consider $F = \{f/2, g/2\}$ and two terms, $f(a,b)$ and $g(a,b)$. Since both terms flatten to $a, b$, the original term cannot be determined based solely on this flat form, because it does not preserve the function symbol. To enable recovery of this information, we encode it in the flattened constraint symbol.

**Definition 6 (Flattening Rules).** *The* flattening rules $\Phi_{f_j}$ *and* $\Phi'$ *replace a given constraint* $c/n$ *with its flat form:*

$$\Phi_{f_j} \ @ \ c(\overline{t_{1,\ldots,i-1}}, t_i, \overline{t_{i+1,\ldots,n}}) \ \texttt{<=>} \ t_i = f_j(t'_1, \ldots, t'_{a_j}) \wedge f_j/a_j \in F \ \texttt{|} \ c_{f_j}(\overline{t_{1,\ldots,i-1}}, \phi(t_i), \overline{t_{i+1,\ldots,n}}) \tag{4.7}$$

$$\Phi' \ @ \ c(\overline{t_{1,\ldots,i-1}}, t_i, \overline{t_{i+1,\ldots,n}}) \ \texttt{<=>} \ t_i = f(t'_1, \ldots, t'_a) \wedge f/a \notin F \ \texttt{|} \ c'(\overline{t_{1,\ldots,i-1}}, \phi(t_i), \overline{t_{i+1,\ldots,n}}) \tag{4.8}$$

Now we can distinguish between the flattened constraints $c_f(a,b)$ and $c_g(a,b)$ by looking at constraint symbols $c_f$ and $c_g$: the former constraint originates from $c(f(a,b))$, and the latter from $c(g(a,b))$.

As each original constraint symbol maps to multiple specialized constraint symbols, rule flattening in case of symbol specialization is more complex than for generic flattening. Every original rule maps to multiple flattened rules, one for each combination of specialized constraint symbols.

**Definition 7 (Flattened Rules).** *The flattening* $\phi$ *of a CHR rule* $(H \ \texttt{?=>} \ G \mid B)$ *with respect to the* $i$*th argument of a constraint* $c/n$ *for the set of function symbols* $F$ *is the set with maximal cardinality, containing rules of the form:*

$$H' \ \texttt{?=>} \ G', G \mid B$$

```
:- chr_constraint c/1, c_f/2, c_g/1, c'/1.

Φ_f @ c(T) <=> T = f(X,Y) | c_f(φ(T)).
Φ_g @ c(T) <=> T = g(Z)   | c_g(φ(T)).
Φ'  @ c(T) <=> T ≠ f(X,Y), T ≠ g(Z) | c'(φ(T)).

r1_f @ c_f(X,Y) \ c_f(X,Y) <=> true.
r1_g @ c_g(Z)   \ c_g(Z)   <=> true.
r1'  @ c'(T)    \ c'(T)    <=> true.
r2_f @ c_f(X,Y) ==> c(X), c(Y).
r3_g @ c_g(Z)   ==> c(Z).
r4_f @ c_f(X,Y) <=> R = ψ_f(X,Y) | write(R).
r4_g @ c_g(Z)   <=> R = ψ_g(Z)   | write(R).
r4'  @ c'(T)    <=> R = ψ'(T)    | write(R).
```

**Table 3.** CHR program from Table 1 after constraint symbol specialization

*where*

- *$H'$ differs from $H$ in that any constraint $c(t_1, \ldots, t_i, \ldots, t_n)$ is replaced by a specialized flattened form, $c_{f_j}(t_1, \ldots, \overline{t'}, \ldots, t_n)$ if $t_i = f_j(t'_1, \ldots, t'_{a_j})$ s.t. $f_j/a_j \in F$, or $c'(t_1, \ldots, t_i, \ldots, t_n)$ otherwise*
- *the new guard $G'$ contains the pre-condition: one $\psi_{f_j}(\overline{t'})$ = $t_i$ for each flattened argument $t_i = f_j(t'_1, \ldots, t'_{a_j})$ s.t. $f_j/a_j \in F$, or $\psi'(t')$ = $t'$ otherwise.*

The flattened program is defined as for generic flattening:

**Definition 8 (Flattened Program).** *The flattening $\phi(P)$ of a CHR program $P$ given by the set of rules $\overline{R}$, with respect to the $i$th argument of a constraint $c/n$, for the set of function symbols $F$, is defined as the flattening of each rule in $\overline{R}$, the flattening and unflattening functions and the flattening rules:*

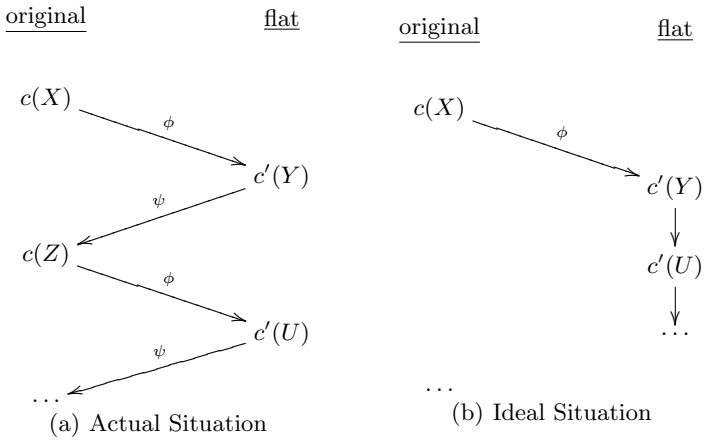
$$\phi(P) = \phi(\overline{R}) \cup \phi \cup \bigcup_{f_j \in F} (\psi_{f_j} \cup \Phi_{f_j}) \cup (\psi' \cup \Phi')$$

*Example 6.* Table 3 shows the program from Table 1 flattened using constraint symbol specialization. The rules $\texttt{r1}_f$ to $\texttt{r3}_g$ have been simplified for readability (see Section 5).

## 5 Post-Processing

The experimental results in Section 6 indicate that the performance improvement attained by better indexing is offset, or in some cases even surpassed, by the run-time overhead of applying the flattening and unflattening functions. In this section we outline two transformations that statically eliminate most of this overhead. The effectiveness of these transformations is borne out by the benchmarks in Section 6.

### 5.1 Repeated Flattening and Unflattening




(a) Actual Situation (b) Ideal Situation

**Fig. 1.** Transitions between original and flattened constraints

Alternating flattening and unflattening of values is a major source of runtime overhead. In a typical scenario (Fig. 1(a)), a value is flattened and matched in a head of a rule, then it is unflattened in that rule's body for calling a new constraint, flattened again to match another rule, and so on. To avoid this overhead, the transformed rules should operate solely on the flattened constraints, whereas the unflattened constraints should be called only by the queries external to the programs.

We propose a four-step rewriting procedure that aims to trigger this ideal scenario (Fig. 1(b)). Execution of a program enhanced with our procedure consists of two phases:

(1) constraint flattening, and
(2) processing of the flattened constraints.

For all but the most trivial programs, we expect the runtime cost of (1) to be marginal with respect to the cost of (2).

We formulate the steps of the procedure in terms of generic flattening; their counterparts for constraint symbol specialization can be easily derived.

**Step 1: Make flattening explicit.**
Unfold constraint calls according to the flattening rules.

*Example 7.* Flattening the rule `d(X,N) <=> N>0, d(X,N-1)` w.r.t. the first argument of `d(X,N)` yields:

```
d'(A,B,N) <=> X = ψ(A,B), N>0 | d(X,N-1).
```

By applying Step 1 to the above rule we obtain:

```
d'(A,B,N) <=> X = ψ(A,B), N>0
            | (A1,B1) = φ(X), d'(A1,B1,N-1).
```

We refer the reader to the work of Tacchella et al. [13] for the formal definition and correctness proof of unfolding of CHR rules.

**Step 2: Eliminate flattening after unflattening.**
Apply the following equation from left to right:

$$\forall \bar{t} : \phi \circ \psi(\bar{t}) = \bar{t}$$

which is valid since $\phi$ is the (left) inverse of $\psi$.

*Example 8.* Applying Step 2 to the last rule in Example 7 yields:
`d'(A,B,N) <=> X = ψ(A,B), N>0 | d'(A,B,N-1).`

**Step 3: Move matchings from unflattened to flattened values.**
Apply the equivalence from left to right:

$$\forall \overline{t_1}, \overline{t_2} : \psi(t_1) = \psi(t_2) \Leftrightarrow \overline{t_1} = \overline{t_2}$$

which is valid since $\psi$ is injective.

*Example 9.* Consider rule `r1` in Table 1. Since the head constraints share the variable `X`, before transformation the rule should be normalized. Flattening the normalized rule yields:

```
c'(A1,B1,C1) \ c'(A2,B2,C2) <=>
        TX = ψ(A1,B1,C1), TY = ψ(A2,B2,C2), TX = TY | true.
```

By applying Step 3, we obtain:

```
c'(A,B,C) \ c'(A,B,C) <=> TX = ψ(A,B,C) | true.
```

**Step 4: Clean up.**
Drop unused unflattening guards and refold the unfolded constraint calls that could not be simplified.

*Example 10.* Applying Step 4 to the last rule in Example 9 yields:
`c'(A,B,C) \ c'(A,B,C) <=> true.`

In general, these rewriting steps are not sufficient to enforce our ideal scenario. However, as the results in Section 6 show, they have good practical effects.

### 5.2 Flattening Indirection

Another source of overhead stems from the processing indirection imposed by flattening, with which all constraints are flattened before the rules are executed. Usually this overhead is marginal. The main exception are, common in CHR, *unconditional simplification rules*—single-headed simplification rules in which the form of a rule's head uniquely determines whether or not that rule is applicable. As a typical example, consider the following fragment of the `zebra` program in our benchmark suite (Section 6):

```
domain(X,[])  <=> fail.
domain(X,[V]) <=> X=V.
domain(X,L1), domain(X,L2) <=> intersect(L1,L2,L), domain(X,L).
```

which after constraint symbol specialization takes the form:

```
domain(X,[]) <=> domain_[](X).
domain(X,[H|T]) <=> domain_[|](X,H,T).

domain_[](_) <=> fail.
...
```

The `domain_[]/1` constraint denotes a base case, which obviously *always*[2] simplifies to `fail`. Because of the flattening indirection, the otherwise short-lived `domain(X,[])` constraints live longer—the liftetime of two calls instead of one. To avoid this indirection overhead we inline the rule body:

```
domain(X,[])    <=> fail.
domain(X,[H|T]) <=> domain_[|](X,H,T).
```

The same technique applies to generic flattening.

## 6 Evaluation

We have implemented our optimizations in two CHR systems on SWI-Prolog: CHRd [9] and K.U.Leuven CHR [14]. All run times, given in seconds for the original programs and relative to the original for the transformed versions, were measured on an Intel Pentium 4, 2.00 GHz, with 512 MB RAM. Our benchmark suite[3] includes several common CHR programs [10]. For each optimization we consider only the relevant benchmarks, for which the transformed programs differ from the original ones.

### 6.1 Generic Flattening

Table 4 shows the results of generic flattening. For each CHR system we list run times measured without flattening or post-processing (−flat,−pp), with flattening but without post-processing (+flat,−pp), and with both flattening and post-processing (+flat,+pp).

In K.U.Leuven CHR generic flattening has little (but mainly positive) effect on all benchmarks, except for `mergesort`, with a speed-up close to 50%, and `gamma_prime`, with almost 40% of slow-down. In CHRd we observe an improvement in `gamma_prime`, `listdom` and `ram`, and a minimal overhead in `mergesort` and `zebra`. For these two small programs, the run-time cost of unflattening exceeds the savings provided by the transformation. All benchmarks demonstrate positive effects of post-processing, and we blame insufficient post-processing for the slow-down of `gamma_prime` in K.U.Leuven CHR: With stronger reasoning on the constraint argument types we (manually) achieved a relative timing of 99%. Hence, further improvement of the automated post-processing seems worthwhile.

[2] Note that the flattening transformation exposes the unconditionality.
[3] Available at `http://www.cs.kuleuven.ac.be/~toms/CHR/Indexing/`

| | K.U.Leuven CHR function symbols | | | CHRd function symbols | | |
|---|---|---|---|---|---|---|
| benchmark | −flat −pp | +flat −pp | +flat +pp | −flat −pp | +flat −pp | +flat +pp |
| gamma_prime | 3.1 | 219% | 137% | 5.4 | 111% | 93% |
| listdom | 5.0 | 114% | 108% | 6.9 | 86% | 84% |
| mergesort | 1.9 | 592% | 56% | 6.6 | 113% | 103% |
| ram_op | 8.8 | 130% | 96% | 8.3 | 94% | 89% |
| ram_prog | 2.9 | 102% | 96% | 4.4 | 91% | 86% |
| zebra | 5.1 | 124% | 93% | 6.4 | 113% | 102% |

**Table 4.** Run times (in sec.) for generic flattening benchmarks

### 6.2 Constraint symbol specialization

Table 5 shows the results of constraint symbol specialization. The columns in the table have the same meaning as in Table 4. Table 5 includes two new benchmarks, `zebra2` and `manners`, not reported in Table 4. The benchmark `zebra2` shows the effect of repeated (until a fixed point is reached) flattening on the `zebra` program: the unoptimized entry in `zebra2` corresponds to the entry in `zebra` processed with the (+flat,+pp) option. The `manners` benchmark involves constraints with constant but no partial-structure arguments, and hence it is not improved by generic flattening. We use this benchmark to demonstrate that constraint symbol specialization may improve the performance of programs without partial structures.

Even before post-processing, constraint symbol specialization has good effects in both systems. In K.U.Leuven CHR only `gamma_prime` and `listdom` suffer performance slow-downs, whereas other benchmarks show run-time improvement. This success is caused by the system's guard optimization [15], which detects dead code for the specialized constraint symbols. Post-processing considerably improves the performance of `listdom` and eliminates the overhead of `gamma_prime`. It has no significant effect on other benchmarks. In CHRd we observe initial performance slow-down in `listdom` and `manners`, the former of which is eliminated by the post-processing step. For `manners`, the cost of processing extra constraints outweighs the benefits of specialization apparent in K.U.Leuven CHR.

In both systems, the repeated flattening of `zebra2` is unsuccessful—its incremental benefit is offset by the incremental overhead.

### 6.3 Improved Time Complexity

Although flattening improves the performance of most benchmarks in our suite, it does not decrease the complexity of the evaluation. We attribute this to the fact that the programmers—aware of CHR's poor handling of partial structures—tend to write already flattened programs, especially for problems which involve referencing partial structure arguments (as in rule (2.1)). Such practice, however, obscures formulation of problems where partial structures appear naturally and

| benchmark | K.U.Leuven CHR function symbols −flat −pp | K.U.Leuven CHR function symbols +flat −pp | K.U.Leuven CHR function symbols +flat +pp | CHRd function symbols −flat −pp | CHRd function symbols +flat −pp | CHRd function symbols +flat +pp |
|---|---|---|---|---|---|---|
| gamma_prime | 1.5 | 114% | 94% | 4.6 | 93% | 83% |
| listdom | 5.2 | 174% | 99% | 7.2 | 129% | 90% |
| manners | 2.2 | 70% | 65% | 4.9 | 131% | 124% |
| mergesort | 7.8 | 30% | 30% | 6.7 | 82% | 81% |
| ram_op | 8.5 | 81% | 82% | 7.5 | 87% | 88% |
| ram_prog | 2.9 | 93% | 93% | 4.5 | 82% | 80% |
| zebra | 5.1 | 96% | 91% | 6.3 | 97% | 97% |
| zebra2 | 4.8 | 103% | 100% | 6.9 | 96% | 97% |

**Table 5.** Run times (in sec.) for constraint symbol specialization benchmarks

are extensively used, e.g., database reasoning. For problems of this kind, flattening does cause complexity improvement, thus extending applicability of CHR to their natural specifications.

For instance, consider a database of employees represented using the constraint `employee(`*Name*`,`*Date*`)`, in which the date of birth *Date* is a compound term `date(`*Day*`,`*Month*`,`*Year*`)`. The following rule finds out which employees' birthdays to celebrate on the current date:

```
check_birthdays(date(Day, Month, CurrentYear)),
        employee(Name, date(Day, Month, YearOfBirth)) ==>
                Age is CurrentYear - YearOfBirth,
                celebrate(Name, Age)
```

The following table lists the run times[4], in milliseconds, before and after flattening the compound date, for three database sizes. The original program exhibits a linear behavior, whereas the run time for the flattened version remains constant.

| program version | number of employees 1,000 | 10,000 | 50,000 |
|---|---|---|---|
| −flat −pp | 2.000 | 22.000 | 108.000 |
| +flat +pp | 0.029 | 0.028 | 0.029 |

The impact of symbol specialization on the birthday program is virtually the same as for generic flattening w.r.t. both the complexity and absolute run times.

### 6.4 Discussion

The results in Tables 4 and 5 suggest that our flattening transformations may improve performance of CHR, however, additional optimizations—such as post-processing—are needed to fully exploit their potential.

Overall, in both systems constraint symbol specialization yields more run-time savings than generic flattening. This, in part, comes from the nature of

[4] in K.U.Leuven CHR; CHRd can evaluate only a transformed version of the rule.

our benchmarks, which do not exhibit the potential of symbol specialization for increasing the program size. The following table shows the number of CHR rules in the original benchmark programs and their two flattened versions:

| benchmarks | original | generic | symbol specialization |
|---|---|---|---|
| `gamma_prime` | 6 | 6 | 8 |
| `listdom` | 14 | 13 | 39 (13) |
| `manners` | 12 | n/a | 14 |
| `mergesort` | 2 | 2 | 3 |
| `ram` | 13 | 13 | 13 |
| `zebra` | 5 | 4 | 4 |

Note that three transformed programs actually contain fewer rules than the corresponding original programs beacuse of inlining of the exposed unconditional simplification rules (see Section 5.2). We see a modest increase in two cases and a considerable increase for the symbol-specialized `listdom` benchmark: from 13 to 39 rules. In case of `listdom`, the blow-up is fully compensated by K.U.Leuven's guard simplification [15]: of the 39 rules, 26 rules have inconsistent guards. In general, however, the increase in program size may be too large to be contained. Such pathological cases must be detected and transformed using generic flattening rather than constraint symbol specialization.

## 7 Related Work

Most relational databases we are aware of do not support compound values. Hence, to map compound data onto (flat) rows requires application of techniques similar to the flattening transformations presented in this section.

*Program Specialization* The need for symbol specialization arises naturally in the context of partial evaluation [16]. Similar, but less ambitious in scope, is the work on constructor specialization for the Glasgow Haskell Compiler [17]. These two approaches, for *single-headed* languages, aim in the first place at reducing intermediate data structures and matching costs, and specializing the body. In contrast, the foremost goal of our approach, for multi-headed CHR rules, is to provide better constraint store indexes. Of course, our techniques benefit from the other effects as well.

*Symbol Indexing Structures* In XSB [18] specialized trie-like structures store previously computed answer substitutions. These substitutions are indexed on their call patterns, and interpreted as partial structure indexes for subsumption-based tabling. However, this approach requires excessive data structure implementation, does not enable further rule specialization, and does not easily compose with other indexes.

The join-calculus [19] features multi-headed rules that are similar in nature to CHR's rules. However, the expressivity of these rules is severly limited: arguments cannot be matched and rules cannot be otherwise guarded. The main

motivation for this limited expressivity is enabled compilation to a highly efficient finite-state automaton [20]. In recent work [21], a flattening specialization similar to ours has been proposed to somewhat lift the severe expressivity restrictions. In this context, the flattening does not improve performance, but rather makes it possible for the rules to be compiled at all.

*CHR Program Transformation* We are not the first to consider program transformation in the context of CHR. Frühwirth [22] proposed the specialization of rules (rather than constraints) with respect to a given goal (rather than head matchings). Tacchella et al. [13] introduced a general technique for unfolding CHR rules in the body of other rules. However, to the best of our knowledge, we are the first to utilize program transformation for performance improvement: we have a fully automated implementation and empirical evidence of its effectiveness. Neither of the above works makes any specific claims about performance improvement, nor demonstrates practical usefulness of the reported technique.

## 8 Conclusion & Future Work

We presented two transformational techniques improving the performance of CHR indexing: generic and specialized function symbol flattening, and a complementary post-processing procedure that compensates their potential overhead.

All techniques have been implemented for the CHRd and K.U.Leuven CHR systems on SWI-Prolog. Evaluation on a set of benchmarks shows that the indexing optimizations enable performance improvement, and that the post-processing is a critical step towards the full realization of their potential. Our approach enables CHR programmers to exploit structured constraint arguments that most naturally fit their applications.

We restrict our attention to function-symbol arguments that are always instantiated. Adding support for uninstantiated arguments is a natural next step in our work. We anticipate this step to further increase the flattening overhead, as well as its complexity when abiding by CHR's refined operational semantics.

The presented framework for CHR program transformation (based on flattening, unflattening and post-processing) is proving useful for expressing other indexing approaches. Our current work concerns a technique for ground terms, called *attributed data* [10], which has better constant factors than hash-tables.

## References

1. Frühwirth, T.: Theory and practice of Constraint Handling Rules. Journal of Logic Programming **37**(1–3) (1998) 95–138
2. Abdennadher, S., Marte, M.: University course timetabling using constraint handling rules. Applied Artificial Intelligence **14**(4) (2000) 311–325
3. Stuckey, P.J., Sulzmann, M.: A Theory of Overloading. ACM Transations on Programming Languages and Systems **27**(6) (2005) 1216–1269
4. Frühwirth, T.: As Time Goes By II: More Automatic Complexity Analysis of Concurrent Rule Programs. Electronic Notes in Theoretical Computer Science **59**(3) (2002)

5. Holzbaur, C., Frühwirth, T.: A Prolog Constraint Handling Rules Compiler and Runtime System. Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules **14**(4) (April 2000)
6. Holzbaur, C., García de la Banda, M., Stuckey, P.J., Duck, G.J.: Optimizing Compilation of Constraint Handling Rules in HAL. Theory and Practice of Logic Programming **5**(Issue 4 & 5) (2005) 503–531
7. Schrijvers, T., Frühwirth, T.: Optimal Union-Find in Constraint Handling Rules. Theory and Practice of Logic Programming **6**(1&2) (2006)
8. Sneyers, J., Schrijvers, T., Demoen, B.: The computational power and complexity of Constraint Handling Rules. In: 2nd Workshop on Constraint Handling Rules (CHR). (2005) 3–17
9. Sarna-Starosta, B., Ramakrishnan, C.: Compiling Constraint Handling Rules for Efficient Tabled Evaluation. In: 9th International Symposium on Practical Aspects of Declarative Languages (PADL). (2007)
10. Sarna-Starosta, B., Schrijvers, T.: Indexing techniques for CHR based on program transformation. Report CW 500, K.U.Leuven, Department of Computer Science (August 2007)
11. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In: 20th International Conference on Logic Programming (ICLP). (2004) 90–104
12. Schrijvers, T., Stuckey, P., Duck, G.: Abstract Interpretation for Constraint Handling Rules. In: 7th International Symposium on Principles and Practice of Declarative Programming (PPDP). (2005)
13. Tacchella, P., Gabbrielli, M., Meo, M.C.: Unfolding in chr. In: 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP). (2007) 179–186
14. Schrijvers, T., Demoen, B.: The K.U.Leuven CHR system: Implementation and application. In: 1st Workshop on Constraint Handling Rules: Selected Contributions. (2004) 1–5
15. Sneyers, J., Schrijvers, T., Demoen, B.: Guard and continuation optimization for occurrence representations of CHR. In: 21st International Conference on Logic Programming (ICLP). (2005) 83–97
16. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)
17. Peyton-Jones, S.: Constructor Specialization for Haskell Programs. In: 12th International Conference on Functional Programming (ICFP). (2007)
18. Warren, D.S., et al.: The XSB Programmer's Manual: version 2.7, vols. 1 and 2 (January 2005) http://xsb.sf.net.
19. Fournet, C., Gonthier, G.: The join calculus: A language for distributed mobile programming. In: International Summer School on Applied Semantics. (2000) 268–332
20. Fessant, F.L., Maranget, L.: Compiling join-patterns. Electronic Notes on Theoretical Computer Science **16**(3) (1998)
21. Ma, Q., Maranget, L.: Compiling pattern matching in join-patterns. In: 15th International Conference on Concurrency Theory (CONCUR). (2004) 417–431
22. Frühwirth, T.: Specialization of Concurrent Guarded Multi-Set Transformation Rules. In: Logic Based Program Synthesis and Transformation (LOPSTR), Revised Selected Papers. (2005)

# Towards Term Rewriting Systems in Constraint Handling Rules

## Coming to terms with jungles

Frank Raiser and Thom Frühwirth

Faculty of Engineering and Computer Sciences, University of Ulm, Germany
{Frank.Raiser|Thom.Fruehwirth}@uni-ulm.de

**Abstract.** Term rewriting systems are a formalism in widespread use, often implemented by means of term graph rewriting. In this work we present preliminary results towards an elegant embedding of term graph rewriting in Constraint Handling Rules with rule priorities ($\text{CHR}^{rp}$). As term graph rewriting is well-known to be incomplete with respect to term rewriting, we aim for sound jungle evaluation in $\text{CHR}^{rp}$. Having such an embedding available allows to benefit from CHR's online property and parallelization potential.

## 1 Introduction

Term rewriting is an important branch of computer science with applications in algebra, recursion theory, software engineering, and programming languages [1]. There is a wealth of known results available concerning term rewriting systems (TRSs).

Constraint handling rules (CHR) is a concurrent committed-choice constraint logic programming language consisting of guarded rules, which transform multisets of atomic formulas (constraints) into simpler ones until exhaustion [2]. Initially created for the development of constraint solvers [3] it has meanwhile grown to a general-purpose programming language [4, 5].

As CHR shares the basic property with TRSs of replacing left-hand sides by right-hand sides, several properties of TRSs have also been investigated in the context of CHR. The most important of these being confluence and termination. However, up to now there is no existing work on embedding a TRS in CHR. Despite their similarities there is a major difference in the way a TRS and a CHR program work, which makes this embedding non-trivial: a TRS can replace subterms of a term independent of how deeply nested the subterm is, whereas CHR replaces multisets of top-level constraints.

Many practical implementations of term rewriting actually perform term graph rewriting [6] instead, which is a sound, but incomplete, alternative to pure term rewriting. The lack of completeness is made up for with the efficiency of the rewriting process. As term graph rewriting can perform multiple term rewrite steps in one step there are even examples of exponential speedups, like the computation of Fibonacci numbers [7]. The reason for these speedups is that

term graph rewriting makes use of structure sharing, such that equal subterms only exist once in a term graph and the need for equality checking, therefore, is avoided. Considering, that term graph rewriting is based on graph transformations for which an embedding in CHR exists [8], this work focuses directly on embedding term graph rewriting into CHR as a means to achieve sound term rewriting.

The theory for term graph rewriting is based on jungles which are introduced in Sect. 2. We also introduce CHR$^{rp}$ [9] there, which is a variant of CHR assigning priorities to rules. It is used in this work instead of plain CHR, as it greatly simplifies the process of updating structure sharing in term graphs without being as restrictive as the refined semantics for CHR. Section 2 further details the correspondence between term rewriting and term graph rewriting, before Sect. 3 presents our approach to embed term graphs with structure sharing in CHR$^{rp}$. It is shown there, that our proposed CHR$^{rp}$ encoding of term graphs ensures a terminating and confluent computation of term graphs with a maximal amount of shared structures. We plan to use these normal form term graphs as a basis for performing term graph rewriting in CHR$^{rp}$, which is outlined in Sect. 4. In that section future work regarding jungle evaluation and properties of the CHR$^{rp}$ implementation of term graph rewriting are outlined as well, before Sect. 5 concludes this work.

## 2 Preliminaries

The following preliminaries are taken from [7]:

**Strings** $A^*$ denotes the set of all strings over some set $A$, including the empty string $\varepsilon$. $f^* : A^* \to B^*$ denotes the homomorphic extension of a function $f : A \to B$.

**Abstract Reductions** Let $\to$ be a binary relation on some set $A$.

We write $\to^+$ and $\to^*$ for the transitive and transitive-reflexive closure of $\to$, respectively. The $n$-fold composition of $\to$ ($n \geq 0$) is denoted by $\to^n$; in particular, $\to^0$ is the equality on $A$.

Some $a \in A$ is a *normal form* (w.r.t. $\to$) if there is no $b \in A$ with $a \to b$. A normal form $a$ is called a *normal form of* $b \in A$ if $b \to^* a$.

We say $\to$ is *terminating* if there is no infinite chain $a_0 \to a_1 \to a_2 \to \ldots$

The relation $\to$ is *confluent* if for all $a, b_1, b_2 \in A$, ${b_1}^* \leftarrow a \to^* b_2$ implies $b_1 \to^* c^* \leftarrow b_2$ for some $c \in A$.

**Terms and Substitution** $T(X)$ denotes the set of all terms over the set $X$ of variables. Terms can contain *function symbols* from the set $\Sigma$. Each function symbol $f$ is associated with an arity $\texttt{arity}(f) \geq 0$. A function symbol $c$ with $\texttt{arity}(c) = 0$ is called a *constant*.

A *substitution* $\sigma : T(X) \to T(Y)$, *rewrite rules*, and *term rewriting systems* are defined as usual.

### 2.1 Constraint Handling Rules with Rule Priorities

This section presents the syntax and operational semantics of constraint handling rules [2, 3]. Constraints are first-order predicates which we separate into *built-in constraints* and *user-defined constraints*. Built-in constraints are provided by the constraint solver while user-defined constraints are defined by a CHR program. A CHR program consists of CHR rules, for which three variants exist: simplification, propagation, and simpagation rules. As simpagation rules are the most general and can directly simulate the other two variants we consider only simpagation rules in this work.

There are different operational semantics available for CHR [4]. We chose to use CHR with rule priorities ($\text{CHR}^{rp}$) for this work, as it is most suitable to the underlying idea of establishing maximal term structure sharing before applying term graph rules. The remaining operational semantics are not as suitable for our work:

**refined semantics** The operational semantics found in most common CHR implementations is the so-called *refined semantics*. It is geared towards implementation issues and its major drawback in our case is that the application order of rules is fixed by their order of occurrence in the program text. In order to be able to generally embed term graph rewriting, however, we require a non-deterministic rule selection as it is available for term graph rewriting.

**abstract semantics** The abstract (or standard) operational semantics is the default operational semantics for CHR, which includes non-deterministic rule selection. However, as we want to ensure, that term graphs use maximal structure sharing before term graph rules are applied to them, additional effort would be required: it has to be guaranteed, that despite the non-deterministic rule selection term graph rules can only be applied after the corresponding term graphs provide for maximal structure sharing.

$\text{CHR}^{rp}$ extends the abstract semantics with priorities for rules, such that rules with the same priority are still selected non-deterministically, but only when no other rules of higher priority can be applied. This allows us to split our rules for the term graph embedding into two classes: a high-priority class of rules responsible for ensuring maximal structure sharing and a low-priority class of rules corresponding to the embedded term graph rewriting rules.

In $\text{CHR}^{rp}$ simpagation rules are of the form

$$\mathit{priority} :: \mathit{Rulename} \ @ \ H_1 \setminus H_2 \Leftrightarrow g \mid C$$

where *priority* is the *priority* of the rule, *Rulename* is an optional unique *identifier* of a rule, the *head* $H_1 \setminus H_2$ is a non-empty conjunction of user-defined constraints, the *guard* $g$ is a conjunction of built-in constraints and the *body* $C$ is a conjunction of built-in and user-defined constraints. Note that with respect to $H_1, H_2$, and $C$ we mix the use of the terms conjunction, sequence, and multiset.

The operational semantics is based on an underlying *constraint theory* $\mathcal{D}$ for the built-in constraints and a *state*, which is a pair $\langle G, S, B, T \rangle$ where $G$ is a *goal*,

i.e. a multiset of user-defined and built-in constraints, $S$ is the CHR *constraint store*, $B$ is the *built-in store*, and $T$ is the propagation history [4]. Table 1 shows the possible state transitions for CHR$^{rp}$ under the operational semantics of CHR with rule priorities, denoted as $\omega_p$.

1. **Solve** $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle G, S, c \wedge B, T\rangle_n$ where $c$ is a built-in constraint.
2. **Introduce** $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle G, \{c\#n\} \cup S, B, T\rangle_{n+1}$ where $c$ is a CHR constraint.
3. **Apply** $\langle \emptyset, H_1 \cup H_2 \cup S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle \Theta(C), H_1 \cup S, B, T \cup \{t\}\rangle_n$ where $P$ contains a rule of priority $p$ of the form

$$p :: r \ @ \ H_1' \setminus H_2' \Leftrightarrow g \mid C$$

and a matching substitution $\Theta$ such that $\mathtt{chr}(H_1) = \Theta(H_1'), \mathtt{chr}(H_2) = \Theta(H_2'), \mathcal{D} \models B \rightarrow \exists_B(\Theta \wedge g)$, $\Theta(p)$ is a ground arithmetic expression and $t = \langle r, \mathtt{id}(H_1) + + \mathtt{id}(H_2)\rangle \notin T$. Furthermore, no rule of priority $p'$ and substitution $\Theta'$ exists with $\Theta'(p') < \Theta(p)$ for which the above conditions hold.

**Table 1.** Transitions of $\omega_p$

### 2.2 Jungle Evaluation and Term Rewriting

A survey on term graph rewriting can be found in [6], with additional details, especially considering jungle evaluation, in [7]. The following definitions and facts are taken from those two works.

It is well-known, how a term can be represented as a tree. The sharing of equal subterms, however, is not allowed in the usual tree structure. To this end, jungles are used, which are a specialization of hypergraphs:

**Definition 1 (Hypergraph).** *A* hypergraph $G = (V_G, E_G, \mathtt{att}_G, \mathtt{lab}_G)$ *consists of a finite set* $V_G$ *of* nodes*, a finite set* $E_G$ *of* hyperedges *(or* edges *for short), and a mapping* $\mathtt{lab}_G : E_G \rightarrow \Sigma$*, labeling hyperedges with function symbols and a mapping* $\mathtt{att}_G : E_G \rightarrow V_G^+$ *such that* $|\mathtt{att}_G(e)| = 1 + \mathtt{arity}(\mathtt{lab}_G(e))$.

*Given* $e \in E_G$ *with* $\mathtt{att}_G(e) = v_0 v_1 \ldots v_n$, $\mathtt{res}(e) = v_0$ *is called the* result node *and* $\mathtt{arg}(e) = v_1, \ldots, v_n$ *are called the* argument nodes*. We define* $\mathtt{indegree}_G(v) = |\{e \mid v \in \arg(e)\}|$ *and* $\mathtt{outdegree}_G(v) = |\{e \mid v = \mathtt{res}(e)\}|$

*Let* $v_1, v_2$ *be two nodes in a hypergraph* $G$*. Then* $v_1 >_G v_2$ *denotes that there is a non-empty path from* $v_1$ *to* $v_2$ *in* $G$*;* $v_1 \geq_G v_2$ *means* $v_1 >_G v_2$ *or* $v_1 = v_2$*.* $G$ *is* acyclic *if there is no node* $v \in V_G$ *such that* $v >_G v$*.*

This allows us to define a jungle:

**Definition 2 (Jungle).** *A hypergraph* $G = (V_G, E_G, \mathtt{att}_G, \mathtt{lab}_G)$ *is a* jungle *if*

1. $\mathtt{outdegree}_G(v) \leq 1 \; \forall v \in V_G$,
2. $G$ *is acyclic.*

When we consider ground terms represented as trees, then all leafs are constants. For jungles these constants become hyperedges with arity 1, i.e. hyperedges which are attached to a result node, but have no argument nodes. Conversely, if a node in a jungle is not a result node of an edge we treat it like a variable. It is easy to see that non-linear terms of a term rewriting system, i.e. terms in which a variable occurs multiple times, can be represented by jungles with one shared variable node per variable of the TRS.

**Notation** $\mathrm{VAR}_G = \{v \in V_G \mid \mathtt{outdegree}_G(v) = 0\}$ denotes the set of variables associated with a jungle $G$.

*Example 1.* Figure 1 shows an exemplary jungle used in a rule for computing Fibonacci numbers. Nodes are shown as black dots and hyperedges as rectangles. For an edge, the associated operation symbol is written inside the rectangle and the result node is given by a line without an arrow, whereas the argument nodes are given by arrows. In general, we assume that the order of arguments coincides with the left-to-right order of arrows in a figure.

In Fig. 1 the node $r$ is a root node of the jungle and the node $n$ is a variable node.

Also see the jungles in Fig. 3 for how jungles allow the sharing of common substructures.

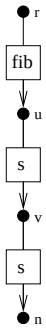



**Fig. 1.** Exemplary jungle $G$

To associate jungles with terms we define a mapping assigning terms to each node of a jungle:

**Definition 3 (Term Representation Function).** *Let $G$ be a jungle. Then*

$$\mathtt{term}_G(v) = \begin{cases} v & \text{if } v \in \mathrm{VAR}_G, \\ \mathtt{lab}_G(e)(\mathtt{term}_G(v_1), \ldots, \mathtt{term}_G(v_n)) & \text{for the unique edge } e \text{ such} \\ & \text{that } \mathtt{att}_G(e) = vv_1 \ldots v_n \end{cases}$$

*defines a function* $\mathtt{term}_G : V_G \to T(\mathrm{VAR}_G)$.

*The set* $\mathtt{term}_G(V_G)$ *of all terms represented by a jungle $G$ is denoted by* $\mathrm{TERM}_G$.

*Example 2.* The terms represented by Fig. 1 are:

| node | $\mathtt{term}_G$ |
|---|---|
| $r$ | `fib(s(s(n)))` |
| $u$ | `s(s(n))` |
| $v$ | `s(n)` |
| $n$ | `n` |

All terms represented by $G$, hence, are:
$\mathrm{TERM}_G = \{\mathtt{fib(s(s(n)))}, \mathtt{s(s(n))}, \mathtt{s(n)}, \mathtt{n}\}$

For the remainder of this work we require various morphisms between jungles, which have to satisfy the following definition:

**Definition 4 (Jungle Morphism).** *Let $G, H$ be jungles. A* jungle morphism *$f : G \to H$ is a pair of mappings $f = (f_V : V_G \to V_H, f_E : E_G \to E_H)$ which preserves sources, targets, and labels, i.e. $\mathtt{att}_H \circ f_E = f_V^* \circ \mathtt{att}_G$ and $\mathtt{lab}_H \circ f_V = \mathtt{lab}_G$.*

*A jungle morphism $f = (f_V, f_E)$ is injective (surjective) if and only if $f_V$ and $f_E$ are both injective (surjective).*

**Notation** $\mathrm{ROOT}_G = \{v \in V_G \mid \mathtt{indegree}_G(v) = 0\}$ denotes the set of *roots* of a jungle $G$.

Analogous to [6] we equate a node $v$ with the set of paths from a root node to $v$ in order to get standard term graphs and avoid the usual isomorphism details. As we allow for multiple root nodes only the path from a specific root node to $v$ is unique, and thus, we include paths from all root nodes to get a unique standard term graph.

As jungles can contain variables, which are represented as nodes that are not result nodes of an edge, every jungle morphism assigning such a node to a node in the target jungle induces a substitution:

**Definition 5 (Induced Substitution).** *Let $f : G \to H$ be a jungle morphism. Then the* induced substitution *$\sigma : T(\mathrm{VAR}_G) \to T(\mathrm{VAR}_H)$ is defined for all $x \in \mathrm{VAR}_G$ by*

$$\sigma(x) = \mathtt{term}_H(f_V(x))$$

*Example 3.* Figure 2 shows a jungle morphism $g$ between two jungles $G$ and $H$. The morphism is depicted by dotted arrows. The jungle $H$ represents the ground term `fib(s(s(s(0))))` which is used to compute the third Fibonacci number. The jungle morphism $g$, which maps $n$ to $g(n) = n'$, induces a substitution $\sigma$ with $\sigma(n) = \mathtt{term}_H(n') = \mathtt{s(0)}$.

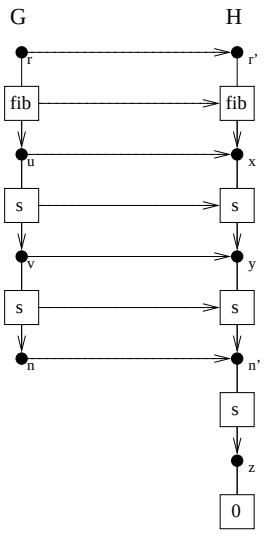



**Fig. 2.** Jungle morphism $g : G \to H$

## 3 Structure sharing in CHR$^{rp}$

This section explains how to encode jungles in CHR$^{rp}$ and introduces a set of rules which implement structure sharing on these jungles. It is shown, that the rules ensure that the maximal amount of structures is shared.

### 3.1 Jungle Encoding in CHR$^{rp}$

**Definition 6 (Jungle Encoding).** *Let $G = (V_G, E_G, \mathtt{att}_G, \mathtt{lab}_G)$ be a jungle. Then $G$ is encoded in CHR$^{rp}$ as follows:*

1. *for all $v \in V_G$ introduce a unique variable $X_v$.*
2. *For each edge $e \in E_G$ with $\mathtt{res}(e) = v$ and $\mathtt{arg}(e) = v_1, \ldots, v_n$ add the constraint $\mathtt{Eq}(X_v, \mathtt{lab}_G(e)(X_{v_1}, \ldots, X_{v_n}))$*

*Let $\mathtt{encode}(G)$ denote the set of $\mathtt{Eq}$ constraints for the CHR$^{rp}$ encoding of $G$ and let $X_{V_G}$ denote the set of variables introduced for the encoding of $G$*[1]*.*

*Let $X_v$ be a variable used in $\mathtt{encode}(G)$. Then*

$$\mathtt{term}(X_v) = \begin{cases} v & \text{if } \nexists\, \mathtt{Eq}(X_v, \ldots) \in \mathtt{encode}(G) \\ op(\mathtt{term}(X_{v_1}), \ldots, \mathtt{term}(X_{v_k}))) & \text{if } \exists\, \mathtt{Eq}(X_v, op(X_{v_1}, \ldots, X_{v_k})) \\ & \quad \in \mathtt{encode}(G) \end{cases}$$

*defines a function $\mathtt{term} : X_{V_G} \to T(X_{V_G})$.*

[1] Note that for each variable $X_v \in X_{V_G}$ there is at most one $\mathtt{Eq}(X_v, \_) \in \mathtt{encode}(G)$ due to $\mathtt{outdegree}(v) \leq 1$ (Def. 2)

*Example 4.* Consider again the jungle $H$ from Fig. 2. It's encoding in CHR$^{rp}$ is: $\texttt{Eq}(X_{r'}, \texttt{fib}(X_x)), \texttt{Eq}(X_x, \texttt{s}(X_y)), \texttt{Eq}(X_y, \texttt{s}(X_{n'})), \texttt{Eq}(X_{n'}, \texttt{s}(X_z)), \texttt{Eq}(X_z, \texttt{0})$.

The following lemma ensures, that the set of terms represented by $\texttt{encode}(G)$ via $\texttt{term}$ is the same as the set of terms represented by $G$ via $\texttt{term}_G$:

**Lemma 1 (Encoding preserves terms).** *For an encoding* $\texttt{encode}(G)$ *of a jungle* $G = (V_G, E_G, \texttt{att}_G, \texttt{lab}_G)$ *it holds that:*

$$\forall X \in X_{V_G} : \texttt{term}(X) \in \text{TERM}_G \tag{1}$$

$$\forall t \in \text{TERM}_G \ \exists X \in X_{V_G} : t = \texttt{term}(X) \tag{2}$$

*Proof. Proof for (1) by structural induction:*
*if* $\nexists\, \texttt{Eq}(X_v, \ldots) \in \texttt{encode}(G)$ *this implies by Definition 6(1) that* $X_v$ *corresponds to a node* $v \in \text{VAR}_G$*, and thus,* $\texttt{term}(X_v) = v = \texttt{term}_G(v) \in \text{TERM}_G$*.*
*if* $\exists\, \texttt{Eq}(X_v, op(X_{v_1}, \ldots, X_{v_k})) \in \texttt{encode}(G)$ *this implies the existence of an edge* $e \in E_G$ *with* $res_G(e) = v$*,* $\texttt{lab}_G(e) = op$ *and* $arg_G(e) = v_1, \ldots, v_k$*. The term* $op(\texttt{term}(X_{v_1}), \ldots, \texttt{term}(X_{v_k}))$*, thus, equals the term (Def. 3)* $\texttt{lab}_G(e)(\texttt{term}_G(v_1), \ldots, \texttt{term}_G(v_k))$ *and is, therefore, contained in* $\text{TERM}_G$*.*

*Each term in* $\text{TERM}_G$ *corresponds to a node* $v \in V_G$ *to which a variable* $X_v \in X_{V_G}$ *is associated. Hence, the proof of (2) is another structural induction analogous to the above.* □

### 3.2 Structure sharing

The idea of structure sharing is that identical subterms can share the same nodes and edges in a jungle. This cuts down on the space usage of an encoded term, as well as allowing to apply a term rewriting rule to all occurrences of the shared subterm in one step. Based on a lemma from [7] this leads us to the basic idea of how to embed term graph rewriting in CHR$^{rp}$: Every jungle $G$ is first transformed into a jungle $\overline{G}$ representing the same terms, but for which its subterm structures are maximally shared. It is then known, that for each application of a term graph rewriting rule to the jungle $G$ the rule also applies to the transformed jungle $\overline{G}$.

Using this property of structure sharing we can provide for a CHR$^{rp}$ embedding which avoids the previously mentioned problem of having to detect subterm equality. Whenever two subterms are equal this is trivially seen from the corresponding jungle nodes and edges being shared. The remaining part of this section, therefore, details how structure sharing can be enforced in CHR$^{rp}$, with the following definitions being adapted from [6]:

**Definition 7 (Collapsing).** *Given two jungles* $G$ *and* $H$*,* $G$ collapses *to* $H$ *if there is a jungle morphism* $f : G \to H$ *such that* $f_V(\text{ROOT}_G) = \text{ROOT}_H$ *and* $\texttt{term}_G(f_V(\text{ROOT}_G)) = \texttt{term}_H(\text{ROOT}_H))$*. This is denoted by* $G \succeq H$*, or if the morphism is non-injective, by* $G \succ H$*. The latter kind of collapsing is said to be* proper*.*

Given the notion of collapsing jungles we can further identify the jungles which are fully collapsed, i.e. to which no more proper collapsing steps are applicable:

**Definition 8 (Tree, Fully Collapsed).** *A jungle $G$ is a* tree *if there is no $H$ with $G \prec H$, while $G$ is* fully collapsed *if there is no $H$ with $G \succ H$.*

The following alternative definition of a fully collapsed jungle is given in [7] and is independent from the notion of collapsing via a jungle morphism:

**Definition 9 (Fully Collapsed, Alternative Definition).** *A jungle $G$ is called* fully collapsed *if* $\texttt{term}_G$ *is injective.*

*Example 5.* Figure 3 shows a jungle which occurs during the computation of $fib(3)$ representing the recursive computation $fib(3) = (fib(1)+fib(0))+fib(1)$. The duplicate occurrence of $fib(1)$ can then be optimized by structure sharing. The collapse step shown in Fig. 3 eliminates one hyperedge representing the term `fib(1)` by reusing another hyperedge which represents the same term. The corresponding jungle morphism is the identity morphism, except for the mapping depicted by the dashed arrows. Overall, structure sharing leads to a linear computation of Fibonacci numbers, as opposed to the naive exponential computation.

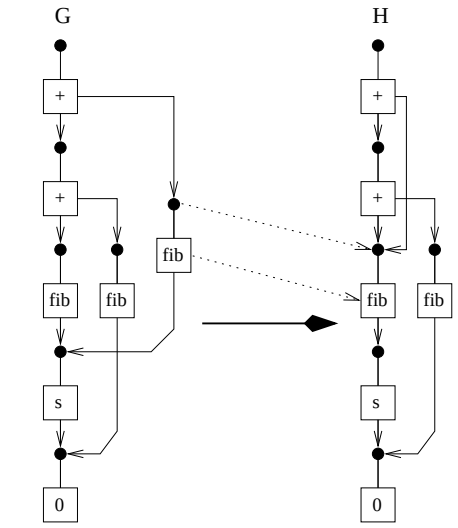



**Fig. 3.** Collapsing of a jungle

The process of collapsing a jungle (called *Folding* in [7]) is instrumented via a set of folding rules according to the following definition:

**Definition 10 (Folding Rule).** *Let $op \in \Sigma$ be a function symbol such that* $\texttt{arity}(op) = k \geq 0$.

*The* folding rule *for op is given by a pair $(L \hookleftarrow K \xrightarrow{b} R)$ of jungle morphisms as depicted in Fig. 4 ("x = y" indicates that b identifies the roots of K; note that L and R have no variables if op is a constant, $\hookleftarrow$ denotes an inclusion morphism). $\mathcal{F}$ denotes the set of folding rules for $\Sigma$.*

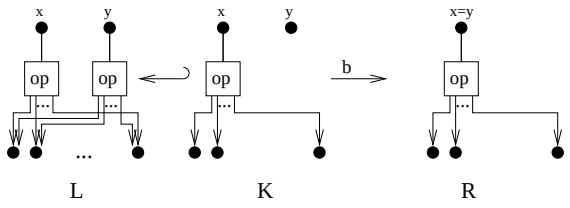



**Fig. 4.** Folding rule for $op \in \Sigma$

We now port this instrumentation to jungles encoded in CHR by `encode`. The generated folding rules are assigned a static priority of 1. This enforces our basic idea of fully collapsing a jungle, before applying rules of the actual TRS to it.

**Definition 11 (CHR folding rule).** *Let $op \in \Sigma$ be a function symbol such that* $\texttt{arity}(op) = k \geq 0$. *Then we define the following CHR folding rule:*

$$1 :: fold\_op \ @ \ \texttt{Eq}(X, op(X_1, \ldots, X_k)) \setminus \texttt{Eq}(Y, op(X_1, \ldots, X_k)) \Leftrightarrow X = Y.$$

*$\mathcal{P}^{\mathcal{F}}$ denotes the CHR program consisting of all CHR folding rules for $\Sigma$.*

*Note that due to CHR's built-in support for syntactic equivalence we can also use the following single fold rule instead:*

$$1 :: fold \ @ \ \texttt{Eq}(X, Term) \setminus \texttt{Eq}(Y, Term) \Leftrightarrow X = Y$$

A single folding step is defined in [7] as follows. In CHR such a folding step coincides with the application of a folding rule, as the following theorem shows.

**Definition 12 (Folding Step).** *Let G be a jungle. A* folding step $G \underset{\mathcal{F}}{\Rightarrow} H$ *from G to a hypergraph H is constructed as follows:*

- *Find a morphism $g : L \to G$ for some folding rule $(L \hookleftarrow K \xrightarrow{b} R)$ such that $g_E$ is injective.*
- *Obtain D from G by removing $g_E(e)$, where e is the unique edge in $L \setminus K$.*
- *Obtain H from D by identifying $g_V(x)$ and $g_V(y)$, where x and y are the roots in L.*

**Fact 1 (Folding Steps Preserve Jungles [7])** *Given a jungle $G$ and a folding step $G \underset{\mathcal{F}}{\Rightarrow} H$, $H$ is a jungle, too.*

**Theorem 1 (CHR folding is sound and complete).** *For a $op \in \Sigma$ and a jungle $G$ the following steps are equivalent:*

1. $G \underset{\mathcal{F}}{\Rightarrow} H$
2. $\texttt{encode}(G) \overset{\omega_p}{\rightarrowtail}_{\mathcal{P}^{\mathcal{F}}} \texttt{encode}(H)$

*Proof.* (1) ⇒ (2)*:*
*Let $g : L \to G$ be the required morphism for the folding rule corresponding to the function symbol $op \in \Sigma$ with $g_E$ being injective. This morphism extends to a matching for the head of the corresponding CHR folding rule for $op$. The two edges in $L$ directly correspond to the two* Eq *constraints in the head of the CHR rule. As $g_E$ is injective there exist two edges $e_1$ and $e_2$ in $g_E(E_L)$ with $\texttt{lab}_G(e_1) = \texttt{lab}_G(e_2) = op$, $\texttt{res}_G(e_1) = v, \texttt{res}_G(e_2) = w$, and $\texttt{att}_G(e_1) = \texttt{att}_G(e_2) = v_1, \ldots, v_k$. By Definition 6 there also exist corresponding constraints $\texttt{Eq}(X_v, op(X_{v_1}, \ldots, X_{v_k}))$ and $\texttt{Eq}(X_w, op(X_{v_1}, \ldots, X_{v_k}))$. Therefore, these two constraints match the two* Eq *constraints in the head of the corresponding CHR folding rule and the rule can be applied.*

*$D$ is obtained from $G$ by removing $g_E(e)$ with $e$ being the unique edge in $L \backslash K$. Let w.l.o.g. $g_E(e) = e_2$ and $X_w = Y$ be the substitution used for the matching of the CHR folding rule's head. Then the application of the simpagation rule removes the* Eq *constraint corresponding to $e_2$, as demanded for the generation of $D$.*

*Finally, $H$ is obtained by identifying $g_V(x)$ and $g_V(y)$ where $x$ and $y$ are the roots in $L$. As defined by the edges $e_1$ and $e_2$ it follows that $g_V(x) = v$ and $g_V(y) = w$. By the implied matching the variables $X$ and $Y$ in the head of the CHR folding rule are bound to the variables $X_v$ and $X_w$. Therefore, the application of the rule unifies $X_v$ with $X_w$ as required by Definition 12.*
(2) ⇒ (1)*:*
*This proof is mostly analogous to the previous argumentation: The CHR matching induces the required morphism $g$ where the injectivity is guaranteed by the multiset semantics of CHR. Additionally, we have to show that applying a CHR folding rule actually results in a state which is an encoding of a jungle. This can, however, be seen from what such a rule does. The encoded graph has to contain two edges with the same label and argument nodes and different result nodes. As one of these edges is removed and it's result node identified with the result node of the other edge the result is again a jungle with the resulting state being its encoding modulo variable renaming.* □

*Example 6.* Consider again the folding step depicted in Fig. 3 and let
$\texttt{encode}(G) = \texttt{Eq}(X_r, +(X_{v_1}, X_{v_2})),$
$\texttt{Eq}(X_{v_1}, +(X_{v_3}, X_{v_4})), \texttt{Eq}(X_{v_2}, \texttt{fib}(X_{v_5})),$
$\texttt{Eq}(X_{v_3}, \texttt{fib}(X_{v_5})), \texttt{Eq}(X_{v_4}, \texttt{fib}(X_{v_6})),$
$\texttt{Eq}(X_{v_5}, \texttt{s}(X_{v_6})), \texttt{Eq}(X_{v_6}, \texttt{0}).$

The CHR folding rule is defined as
$1 :: fold \ @ \ \texttt{Eq}(X, Term) \setminus \texttt{Eq}(Y, Term) \Leftrightarrow X = Y$
and can be applied to the `Eq` constraints for $X_{v_2}$ and $X_{v_3}$ leading to the following CHR state:
$\texttt{encode}(H) = \texttt{Eq}(X_r, +(X_{v_1}, X_{v_2})),$
$\texttt{Eq}(X_{v_1}, +(X_{v_2}, X_{v_4})), \texttt{Eq}(X_{v_2}, \texttt{fib}(X_{v_5})),$
$\texttt{Eq}(X_{v_4}, \texttt{fib}(X_{v_6})), \texttt{Eq}(X_{v_5}, \texttt{s}(X_{v_6})), \texttt{Eq}(X_{v_6}, \texttt{0}).$

Using Theorem 1 several properties of jungle folding can be transferred to $\mathcal{P}^{\mathcal{F}}$:

**Fact 2 (Folding Steps Preserve Terms [7])** *Let $G \underset{\mathcal{F}}{\Rightarrow} H$ be a folding step. Then $\mathrm{TERM}_G = \mathrm{TERM}_H$.*

**Corollary 1 (CHR folding preserves terms).** *The application of a CHR folding rule $\texttt{encode}(G) \overset{\omega_p}{\rightarrowtail}_{\mathcal{P}^{\mathcal{F}}} \texttt{encode}(H)$ preserves the terms represented by the encoded jungle $G$.*

*Proof. A direct consequence of Fact 2 and Theorem 1.* □

**Fact 3 ([7])** $\underset{\mathcal{F}}{\Rightarrow}$ *is terminating and confluent.*

**Corollary 2 (CHR folding is terminating and confluent).** $\overset{\omega_p}{\rightarrowtail}_{\mathcal{P}^{\mathcal{F}}}$ *is terminating and confluent.*

*Proof. This follows directly from the soundness and completeness result in Theorem 1 and Fact 3.* □

Following the idea of fully collapsing jungles by the application of folding rules, we transfer the following fact to CHR:

**Fact 4 ([7])** *A jungle $G$ is fully collapsed if and only if there is no folding step $G \underset{\mathcal{F}}{\Rightarrow} H$.*

**Corollary 3 (fully collapsed with CHR folding).** *Let $G$ be a jungle with encoding $\texttt{encode}(G)$. $G$ is fully collapsed if and only if there is no rule in $\mathcal{P}^{\mathcal{F}}$ applicable to $\texttt{encode}(G)$.*

*Proof. This is a direct consequence of Thm. 1 and Fact 4.*

**Corollary 4 (CHR folding fully collapses).** *Let $G$ be a jungle with encoding $\texttt{encode}(G)$. Then there exists a terminating computation $\texttt{encode}(G) \overset{\omega_p{}^*}{\rightarrowtail}_{\mathcal{P}^{\mathcal{F}}} \texttt{encode}(H)$, such that the jungle $H$ is fully collapsed.*

*Proof. This is a direct consequence of Corollary 2 and Corollary 3.* □

## 4 Discussion and Future Work

Targeting term graph rewriting instead of term rewriting allows us to avoid equivalence problems occurring with non-linear TRSs. A non-linear TRS allows the usage of the same variable multiple times on the left-hand side in order to express equal subterms. Considering a direct approach of flattening a term into a linear number of CHR constraints and associating a variable to each subterm has shown to be problematic in terms of these non-linear equalities.

One possible approach is to compute equality of subterms eagerly, similar to the structure sharing presented in this work. However, when the structures are not shared, but the constraint store only knows that two structures are equal a rewrite rule could change only one of the structures. This leads to the constraint store still modeling equivalence between the structures, and thus, a recomputation is required to regain a consistent store.

Another possibility is to include checking equivalent subterms in guards for non-linear rules resulting in the following kind of rules:

$$c(\ldots, X, \ldots, Y, \ldots), H \Leftrightarrow test\_eq(X, Y) \wedge G \mid B$$

Technically however, these constraints are not built-in, as they have to inspect the constraint store and CHR only allows built-in constraints in guards. Hence, the computation of these equality checks can be triggered by additional propagation rules according to the following scheme:

$$\begin{aligned} c(\ldots, X, \ldots, Y, \ldots), H &\Rightarrow test\_eq(X, Y, R) \\ c(\ldots, X, \ldots, Y, \ldots), test\_eq(X, Y, 1), H &\Leftrightarrow G \mid B \end{aligned}$$

This approach, however, is targeted towards the refined semantics of CHR, as a non-deterministic execution model resembles eager computation – as all propagation rules may fire first leading to the computation of all possible equalities.

Using Adaptive CHR [10] for eagerly computing equivalent subterms is another approach we plan to investigate in the future. With Adaptive CHR the equivalence of subterms also contains a justification, such that in the case of the replacement of a term in only one of the subterms the justification is violated and equivalence is recomputed on demand. While term graph rewriting is incomplete w.r.t. pure term rewriting an embedding in Adaptive CHR can achieve completeness at the cost of the efficiency granted by term graph rewriting.

The collapsing process detailed above is a necessary prerequisite for embedding term graph rewriting in CHR. The next step is the application of jungle evaluation rules to a jungle in order to simulate one or more term rewriting steps. This application is based on general graph transformations using the double pushout approach [11]. An embedding for graph transformations in CHR has already been investigated in [8].

However, the matching of left-hand sides of jungle evaluation rules to host jungles has to be injective in CHR due to its multiset semantics. Consider the jungle evaluation rule in Fig. 5, which represents the term rewriting rule $+(0, 0) \rightarrow 0$. As we take care that host jungles are fully collapsed, the left-hand side of the rule

has to use a non-injective matching for the two 0-edges to a shared occurrence of such a 0-edge.

We plan to investigate the possibility of collapsing the jungles occurring in a jungle evaluation rule in order to enforce an injective matching. Figure 6 shows how the resulting jungle evaluation rule for the rule in Fig. 5 looks like. In CHR this can easily be realized, by using each of the jungles as input to $\mathcal{P}^{\mathcal{F}}$ and use the collapsed output jungle for the construction of the corresponding CHR$^{rp}$ rule. However, additional investigations are required to ensure, that using these collapsed rules is sound and complete with respect to the original rules being applied to fully collapsed jungles.

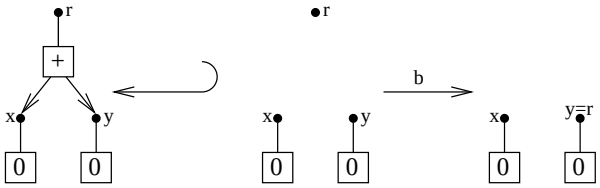



**Fig. 5.** Jungle evaluation rule

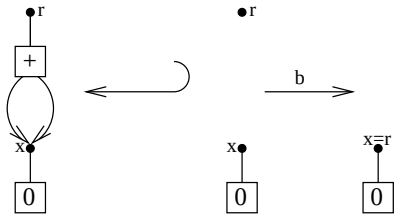



**Fig. 6.** The collapsed jungle evaluation rule

With a guaranteed injective matching the results from [8] can then be reused in order to perform term graph rewriting in CHR$^{rp}$. Slight adjustments will be necessary to account for the possible non-injectivity of $b$ and due to the structure sharing idea no edges – except for the one representing the topmost term which is replaced by the rule – must be removed. This could result in a possibly large proportion of the constraint store being garbage left over from rule applications, and thus, requires the addition of cleanup rules. This garbage problem also conflicts with confluence, which is solved by considering *pointed reductions* in [6]. We expect to get a cleaner result for confluence modulo garbage by applying the results on observable confluence [12] in CHR.

## 5 Conclusion

In this paper we provide a basis for embedding term graph rewriting in Constraint handling rules with rule priorities (CHR$^{rp}$). We presented how jungles

are representing term graphs, and how these jungles can be encoded in CHR$^{rp}$ such that the encoding is term preserving. We then introduced the concepts of structure sharing and fully collapsing a jungle, for which we proposed means to achieve them in a sound and complete, as well as terminating and confluent, way in CHR$^{rp}$. Furthermore, we outlined initial ideas for the remaining part of the embedding of term graph rewriting in CHR$^{rp}$.

## References


1. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press, New York, NY, USA (1998)
2. Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Springer-Verlag (2003)
3. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37**(1-3) (October 1998) 95–138
4. Sneyers, J., Weert, P.V., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules — A survey of CHR research between 1998 and 2007. Submitted to Journal of Theory and Practice of Logic Programming (2008)
5. Frühwirth, T.: Constraint handling rules. to appear (2008)
6. Plump, D.: Term graph rewriting. In: Handbook of Graph Grammars and Computing by Graph Transformations. Volume 1., World Scientific (1997) 3–61
7. Hoffmann, B., Plump, D.: Implementing term rewriting by jungle evaluation. Informatique Théorique et Applications **25** (1991) 445–472
8. Raiser, F.: Graph Transformation Systems in CHR. In Dahl, V., Niemelä, I., eds.: Logic Programming, 23rd International Conference, ICLP 2007. Volume 4670 of Lecture Notes in Computer Science., Porto, Portugal, Springer-Verlag (September 2007) 240–254
9. De Koninck, L., Schrijvers, T., Demoen, B.: User-definable rule priorities for CHR. In: PPDP '07: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming, New York, NY, USA, ACM (2007) 25–36
10. Wolf, A.: Adaptive constraint handling with CHR in java. In: Principles and Practice of Constraint Programming, 7th International Conference, CP 2005. (2001) 256–270
11. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer-Verlag (2006)
12. Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable confluence for constraint handling rules. In Dahl, V., Niemelä, I., eds.: Logic Programming, 23rd International Conference, ICLP 2007. Volume 4670 of Lecture Notes in Computer Science., Porto, Portugal, Springer-Verlag (September 2007) 224–239

# Termination Analysis of CHR revisited

Paolo Pilozzi* and Danny De Schreye

Dept. of Computer Science, K.U.Leuven, Belgium
paolo.pilozzi@cs.kuleuven.be danny.deschreye@cs.kuleuven.be

**Abstract.** Today, there exist two distinct direct approaches to prove termination of CHR programs. Both are applicable on separate classes of CHR programs. One approach, by T. Frühwirth, proves termination of CHR programs without propagation rules. A second approach deals with CHR programs with propagation rules. Due to its extended scope to such programs, it fails to prove termination for a class of CHR programs without propagation that are in the scope of Frühwirth's approach. In this paper, we introduce a termination condition for CHR which is strictly more powerful than both of these and deals with a new class of programs.
Keywords: Constraint Handling Rules, Analysis, Termination.

## 1 Introduction

Constraint Handling Rules (CHR), created by Thom Frühwirth [1], is a declarative programming languages. It is a concurrent, committed-choice, logic programming language. CHR is constraint-based and has guarded rules that rewrite multisets of atomic formulas. Its simple syntax and semantics make it well-suited for implementing arbitrary solvers [1, 2]. It is the latter feature of the language that caused its success and impact on the research community [3–5].

Many solvers were implemented in CHR, some examples can be found in [6,7]. However, to make the language appealing to the larger public, thorough analysis techniques need to be developed to improve both comprehension of and compilation schemes for the language. One of the most important properties of CHR programs is termination. As the execution of multi-headed rules is fairly complicated, mistakes are easily made, leading to unwanted infinite computations. Also, other important properties of CHR programs, such as confluence [8, 9], require that a constraint solver is terminating.

Termination analysis of CHR programs received increasing attention in recent years. The main work in the field was presented some years ago in [10]. There, Frühwirth demonstrates that termination proof methods from Logic Programming (LP) and Term-Rewrite Systems (TRS) are applicable to the CHR context. By measuring the multiset size of the constraint store and formulating conditions on simplification rules, he was able to successfully prove an important class of CHR programs terminating. Unfortunately, propagation rules cause explicit increases. Since such rules only add constraints, a different approach was required for programs with propagation rules.

* Supported by I.W.T. Flanders - Belgium

Recently, a second direct approach was presented in [11]. Beside programs with simplification, it proves termination of programs with propagation. This was possible by considering a fire-once policy for propagation rules [5, 12]. A new termination condition was formulated. The condition compares individual constraints in rules, as such guaranteeing finite addition of constraints when executing a CHR program. However, in the case of a simplification only programs, it is less powerful than the approach for CHR without propagation.

In this paper, we present a new approach to prove termination of general CHR programs. Our approach generalizes both of the existing direct approaches and is able to prove a new class of CHR programs terminating. We develop for this purpose a new measure for CHR states, based on a new concept: the *propagation store*. The latter represents constraints which can be added to a state by only propagating on that state. By also introducing the notion of a token store, to prevent trivial non-termination by propagation rules, we prove decreases on any kind of rule by composing all three stores into a single lexicographical description of a CHR state.

***Overview.*** In a preliminary section, we recall CHR by its syntax and theoretical operational semantics. There, we introduce the concept of a propagation store. The next section discusses termination of CHR programs. First, we introduce some general notions, adapted from termination analysis in other declarative languages. Then, we recall the existing approaches, to be able to later compare them with our approach. Finally, we present our approach, where we first discuss a ranking condition on propagation and afterwards one on simplification. We prove that our conditions are sufficient to prove termination. In Section 4, we discuss our method. Finally, in Section 5, we draw conclusions.

## 2 CHR syntax and semantics

In this section, we recall syntax and semantics of CHR [1, 2, 5, 12] and introduce new concepts w.r.t. semantics, used for termination of CHR programs.

### 2.1 Syntax of CHR

CHR manipulates conjunctions of *constraints*.

**Definition 1 (Constraint).** *A constraint is syntactically defined as a special first-order predicate $c(t_1, \ldots, t_n)$ of arity $n \geq 0$. We distinguish built-in constraints from CHR constraints. Built-in constraints are pre-defined and solved by an underlying constraint solver $CT$. CHR constraints are user-defined and solved by a CHR program $P$.* □

A *CHR program* relates conjunctions of constraints by three different kinds of rules. A *simplification rule* replaces constraints by simpler constraints, a *simpagation rule* replaces constraints conditional on the presence of other constraints and a *propagation rule* adds constraints without removing any.

**Definition 2 (CHR program).** *Let $H_k$, $H_r$ and $C$ denote conjunctions of CHR constraints and let $G$ and $B$ denote conjunctions of built-in constraints. A CHR program $P$ is a finite set of CHR rules of the following form:*

| *Simplification rule:* | *Propagation rule:* | *Simpagation rule:* |
|---|---|---|
| $H_r \Leftrightarrow G \mid B, C.$ | $H_k \Rightarrow G \mid B, C.$ | $H_k \setminus H_r \Leftrightarrow G \mid B, C.$ |

*CHR rules are named by adding "rulename @" in front of the rule.* □

The next example program in CHR computes prime numbers.

*Example 1 (Primes).* The primes example consists of two different kinds of rules:

$$test \ @ \ primes(M) \setminus primes(N) \Leftrightarrow N > M, N \ mod \ M \ is \ 0 \mid true.$$
$$generate \ @ \ primes(N) \Rightarrow N > 2 \mid Np \ is \ N - 1, primes(Np).$$

The first rule, a simpagation rule, tests whether a generated prime number has a divisor. If this is the case, it is removed. The second rule, a propagation rule, generates numbers for prime evaluation, top-down. □

In the following, we refer by simplification to both simplification and simpagation, as these have a similar behavior w.r.t. termination.

### 2.2 The theoretical operational semantics of CHR

A CHR program defines a state transition system, where the transition relation is given by the rules in the program and by the underlying CT for solving built-ins. Declaratively, simplification defines a logical equivalence between removed constraints $H_r$ and added constraints $B \wedge C$, provided that the kept constraints $H_k$ are present in the constraint store and that the guard $G$ holds: $G \rightarrow (H_k \rightarrow H_r \leftrightarrow B \wedge C)$. For propagation rules, there are no removed constraints. Added constraints are therefore a consequence of constraints present in the constraint store: $G \rightarrow (H \rightarrow B \wedge C)$.

Operationally, rules are applied exhaustively on the CHR constraints in the *constraint store*, until an answer state is reached. Rule application is non-deterministic, meaning that we can choose to fire any of the rules applicable to a CHR state. This choice is however a committed choice, it cannot be undone. When solving built-ins, this results either in a failed state or in an introduction of bindings for variables, called a computed answer substitution (c.a.s.).

***Constraint store.*** The constraint store is a collection of labeled CHR constraints and built-in constraints. Simplification replaces constraints with "simpler" ones. Propagation only adds constraints to complete state information.

**Definition 3 (Constraint store).** *The constraint store is a set $S$ of uniquely labeled CHR constraints $c\sharp i$ and built-in constraints $b$. We define $chr(c\sharp i) = c$ to obtain the constraint and $id(c\sharp i) = i$ to obtain the label.* □

Labeling constraints is required to prevent *trivial non-termination*. This kind of non-termination is caused by propagation in the program. Propagation rules do not remove constraints and therefore are infinitely applicable on the same combination of constraints. In order to keep track of which combinations of constraints that have caused which propagation rules to fire, a *fire-once policy* is introduced on combinations of labeled constraints.

***Token store.*** As propagation rules need an applicability condition to avoid trivial non-termination [1, 12], we introduce the token store. This store keeps information about how propagation rules can be applied on a given set of CHR constraints. Once a propagation rule has been applied to these constraints, the appropriate token is removed, so that the rule cannot be reapplied on the same combination of constraints, as such implementing a fire-once policy.

**Definition 4 (Token store).** *Let $P$ be a CHR program and $S$ a constraint store. Then a token store $T$ given $S$, is a set of tokens $(R_i, id_1, \ldots, id_n)$, where $(R_i \ @ \ h_1, \ldots, h_n \Rightarrow G \mid B, C)$ is a propagation rule in $P$ and where $c_j \sharp id_j$ are constraints in $S$, such that $\exists \sigma\theta : CT \models c_1 = h_1\sigma \wedge \cdots \wedge c_n = h_n\sigma \wedge G\sigma\theta$. Here, $\sigma$ is a match substitution for the heads and $\theta$ a c.a.s. for guard satisfaction.* □

Our definition of the token store is a more strict version of the one in [12]. There, constraints have to be unifiable with heads. In such a setting, tokens can appear in the token store, not corresponding to applicable rules. It covers however for propagation rules that become applicable as a consequence of solving built-ins. We disregard such programs. Therefore in our case, tokens are also only added as a consequence of adding CHR constraints.

**Definition 5 (Addition of tokens).** *Let $P$ be a CHR program, $S$ a constraint store and $C_0$ a labeled CHR constraint added to the constraint store $S$, then*

$$T^A_{(C_0,S)} = \{(R, i_1, \ldots, i_n) \mid (R \ @ \ h_1, \ldots, h_n \Rightarrow G \mid B, C) \in P, \text{ where } \{c_1 \sharp i_1, \ldots, c_n \sharp i_n\} \subseteq C_0 \cup S \text{ such that } C_0 \in \{c_1 \sharp i_1, \ldots, c_n \sharp i_n\} \text{ and } CT \models \exists \sigma\theta : (c_1 = h_1\sigma) \wedge \cdots \wedge (c_n = h_n\sigma) \wedge G\sigma\theta$$

*If multiple constraints $C_0 = \{C^1, \ldots, C^n\} = \{c^1 \sharp i^1, \ldots, c^n \sharp i^n\}$ are added, then*

$$T^A_{(C_0,S)} = T^A_{(C^1,S)} \cup T^A_{(C^2, C^1 \cup S)} \cup \cdots \cup T^A_{(C^n, C^1 \cup \cdots \cup C^{n-1} \cup S)}$$

*Note that $\sigma$ is a match substitution and $\theta$ a c.a.s.* □

When removing constraints from the store, the invalid tokens are removed.

**Definition 6 (Elimination of tokens).** *Let $T$ be a token store and $C_0 = \{C^1, \ldots, C^n\}$ labeled CHR constraints removed from the constraint store $S$, then*

$$T^E_{(C_0,T)} = \{(R, i_1, \ldots, i_n) \in T \mid \exists C^j \in C_0 : id(C^j) \in \{i_1, \ldots, i_n\}\}$$ □

Simplification corresponds to $T' = (T \backslash T^E_{(H_r,T)}) \cup T^A_{(C, S \backslash H_r)}$, where $H_r$ are the CHR constraints removed by simplification and $C$ the CHR constraints added. Propagation on the other hand corresponds to $T' = (T \backslash \{(R, i_1, \ldots, i_n)\}) \cup T^A_{(C,S)}$, where $\{(R, i_1, \ldots, i_n)\}$ is the token removed when applying the propagation rule.

***CHR state and CHR transition relation.*** A CHR state is a tuple of two elements: the constraint store and the token store. It is annotated with a fresh integer value, used to label constraints which enter the constraint store.

**Definition 7 (CHR state).** *A CHR state is a tuple $\langle S, T\rangle_\nu$, where $S$ is the constraint store and $T$ the token store. Every state is annotated with a fresh integer $\nu$, not yet assigned to a constraint. An* initial state *is a tuple $\langle S, T^A_{(S,\emptyset)}\rangle_v$, with $v$ a fresh integer value. In a* final state *no more transitions are possible.* □

Transitions between states occur in two different settings. Either at least one constraint is removed and we simplify the state or no constraints are removed and we propagate on the state. The *transition relation* $\rightarrow$ between CHR states, given $CT$ for built-ins and $P$ for CHR constraints, is therefore defined as follows.

**Definition 8 (Transition relation).** *Let $H_k = h_1, \ldots, h_j$, $H_r = h_{j+1}, \ldots, h_n$ and $C = d_1, \ldots, d_m$ denote conjunctions of CHR constraints, let $G$ and $B$ denote conjunctions of built-in constraints, let $\sigma$ be a match substitution for the heads of the rule $R$ and let $\theta$ be a c.a.s. for built-ins in the guard. Then, $\rightarrow$ is:*

| ***Solve*** |
|---|
| ***IF*** $S = b \cup S'$, *where $b$ is a built-in constraint such that* $CT \models \exists\theta : b\theta$<br>***THEN*** $\langle S, T\rangle_\nu \xrightarrow{CT\theta} \langle S'\theta, T\rangle_\nu$ |

| ***Simplify*** |
|---|
| ***IF*** $(R_s \ @\ H_k \setminus H_r \Leftrightarrow G \mid B, C)$ *is a fresh variant of a rule in $P$*<br>*and* $S = H'_k \cup H'_r \cup S'$, *with*<br>$H'_k = \{c_1\sharp i_1, \ldots, c_j\sharp i_j\}$ *and* $H'_r = \{c_{j+1}\sharp i_{j+1}, \ldots, c_n\sharp i_n\}$<br>*such that* $CT \models \exists\sigma\theta : (c_1 = h_1\sigma) \wedge \cdots \wedge (c_n = h_n\sigma) \wedge G\sigma\theta$<br>***THEN*** $\langle S, T\rangle_\nu \xrightarrow{R_s\sigma\theta} \langle S'', T''\rangle_{\nu+m}$<br>*where* $S'' = (H'_k \cup S' \cup B \cup \{d_1\sharp\nu, \ldots, d_m\sharp(\nu+m-1)\})\sigma\theta$ *and*<br>*where* $T'' = ((T \setminus T^E_{(H'_r,T)}) \cup T^A_{(\{d_1\sharp\nu,\ldots,d_m\sharp(\nu+m-1)\},H'_k\cup S')})$ |

| ***Propagate*** |
|---|
| ***IF*** $(R_p \ @\ H_k \Rightarrow G \mid B, C)$ *is a fresh variant of a rule in $P$*<br>*and* $S = H'_k \cup S'$, *with* $H'_k = \{c_1\sharp i_1, \ldots, c_j\sharp i_j\}$<br>*and* $T = \{(R_p, i_1, \ldots, i_j)\} \cup T'$<br>*such that* $CT \models \exists\sigma\theta : (c_1 = h_1\sigma) \wedge \cdots \wedge (c_j = h_j\sigma) \wedge G\sigma\theta$<br>***THEN*** $\langle S, T\rangle_\nu \xrightarrow{R_p\sigma\theta} \langle S'', T''\rangle_{\nu+m}$<br>*where* $S'' = (S \cup B \cup \{d_1\sharp\nu, \ldots, d_m\sharp(\nu+m-1)\})\sigma\theta$ *and*<br>*where* $T'' = (T' \cup T^A_{(\{d_1\sharp\nu,\ldots,d_m\sharp(\nu+m-1)\},S)})$ |

*Note that the label represented by $\nu$ is an integer assigned to the first constraint in $C$ that is added to the constraint store. Then, $\nu + 1$ is assigned to the second added constraint and so on.* □

The definition for the transition relation states that for a rule to be applicable there must exist matching CHR constraints in the constraint store for which the guard can be satisfied.

*Example 2 (Executing Primes).* We revisit Primes from Example 1 and execute it with a query $\langle\{primes(7)\sharp 1\}, \{(R_2, 1)\}\rangle_2$. We get as a possible computation:

$$
\begin{aligned}
I_0 &= \langle\{primes(7)\sharp 1\}, \{(R_2,1)\}\rangle_2 &&\xrightarrow{R_2}\\
I_1 &= \langle\{primes(7)\sharp 1, primes(6)\sharp 2\}, \{(R_2,2)\}\rangle_3 &&\xrightarrow{R_2}\\
I_2 &= \langle\{primes(7)\sharp 1, primes(6)\sharp 2, primes(5)\sharp 3\}, \{(R_2,3)\}\rangle_4 &&\xrightarrow{R_2}\\
I_3 &= \langle\{primes(7)\sharp 1, primes(6)\sharp 2, primes(5)\sharp 3, primes(4)\sharp 4\}, \{(R_2,4)\}\rangle_5 &&\xrightarrow{R_2}\\
I_4 &= \langle\{primes(7)\sharp 1, primes(6)\sharp 2, primes(5)\sharp 3, primes(4)\sharp 4, primes(3)\sharp 5\}, \{(R_2,5)\}\rangle_6 &&\xrightarrow{R_1}\\
I_5 &= \langle\{primes(7)\sharp 1, primes(5)\sharp 3, primes(4)\sharp 4, primes(3)\sharp 5\}, \{(R_2,5)\}\rangle_6 &&\xrightarrow{R_2}\\
I_6 &= \langle\{primes(7)\sharp 1, primes(5)\sharp 3, primes(4)\sharp 4, primes(3)\sharp 5, primes(2)\sharp 6\}, \emptyset\rangle_7 &&\xrightarrow{R_1}\\
I_7 &= \langle\{primes(7)\sharp 1, primes(5)\sharp 3, primes(3)\sharp 5, primes(2)\sharp 6\}, \emptyset\rangle_7
\end{aligned}
$$

Note that we have omitted discussion of the solve transition. We assume in this example that built-ins are solved immediately. □

### 2.3 Solving built-in constraints

Solve transitions in computations introduce bindings. These can be delayed, however. Whether bindings caused by such built-ins are available at the time CHR constraints cause rules to fire, is therefore uncertain. In some cases however, delaying a built-in causes a CHR constraint to be delayed as well. The next example revisits a rule of Primes in Example 1 in which this is the case.

*Example 3. generate* @ $primes(N) \Rightarrow N > 2 \mid Np$ *is* $N - 1, primes(Np)$.
When adding the built-in $Np$ *is* $N - 1$, it can be noted, given the guard $N > 2$, that the added CHR constraint $primes(Np)$ can only fire the rule again when the built-in has been solved. That is, the constraints matching with $primes(N)$ in the head of the rule have to be ground. Therefore, in this case, we can use the built-in to infer interargument relations. □

By analyzing which constraints can fire rules, we know to what extent argument positions will be instantiated. By considering therefore a call set and a rigid interpretation in this call set, we can use built-ins, as we only measure argument positions which will be instantiated enough. As such, only interargument relations from added built-ins of which we know these must introduce bindings, are taken into account. When measuring sizes of added built-ins, we assign the level value 0. After all, we assume them to terminate on their own.

### 2.4 CHR computations

In CHR, a transition is often called a *computation step* and a sequence of computation steps is called a *computation*. For a query $I$ and a program $P$, there are usually several different possible computations.

Termination of CHR programs executed under a theoretical semantics corresponds to the notion of universal termination, where we require finiteness of all computations originating from a query.

**Definition 9 (Termination of a CHR program).** *We say that a CHR program $P$ terminates for a query $I$ iff all computation of $P$ for $I$ are finite.* □

Without loss of generality, we regard computations in CHR as a subsequence of simplification steps, interleaved with sequences of zero or more propagation steps. Computations are therefore of the following form,

$$s_{(1,1)} \xrightarrow{R_{p_{(1,1)}}} s_{(1,2)} \xrightarrow{R_{p_{(1,2)}}} \ldots \xrightarrow{R_{s_1}} s_{(2,1)} \xrightarrow{R_{p_{(2,1)}}} s_{(2,2)} \xrightarrow{R_{p_{(2,2)}}} \ldots \xrightarrow{R_{s_2}} \ldots$$

where $R_{p_{(i,j)}}$ represents the application of a propagation rule and $R_{s_i}$ that of a simplification rule. Note that a propagation sequence may be infinitely long.

### 2.5 The propagation store

Propagation in CHR computations serves as a way to complete state information. When the state after simplification is propagated upon, we reach some more completed state on which we can further simplify.

W.r.t. propagation we distinguish two different kinds of CHR states. A *fully propagated state* is a state with an empty token store. Therefore, no propagation can take place on a fully propagated state. A *partially propagated state* is a state which does contain tokens. We define the action of *full propagation* and *partial propagation*, where we refer to a sequence of propagation steps, originating from a CHR state and ending in a fully or partially propagated state, respectively.

When fully propagating on a state, independent of the order in which propagation rules are applied, we end up in an equivalent state. Such equivalent states contain the same constraints. However, they may be labeled differently as propagation rules are applied in different orders.

Given a state, we refer to the *propagation store* as the multiset of all the constraints that are added when fully propagating on the given state. Obviously, the constraints added by partially propagating on a state correspond to a subset of the propagation store as applying a propagation rule now corresponds to the introduction of constraints from the propagation store to the constraint store. Up to identification, their union therefore remains constant under propagation.

*Example 4 (Executing Primes).* We revisit the computation of Primes from Example 2 and represent for it the propagation store:

$$\begin{aligned}
&\langle\{primes(7)\sharp 1\}, \{\mathbf{primes(6)}, \mathbf{primes(5)}, \mathbf{primes(4)}, \mathbf{primes(3)}, \mathbf{primes(2)}\}, \{(R_2, 1)\}\rangle_2\\
&\langle\{primes(7)\sharp 1, primes(6)\sharp 2\}, \{\mathbf{primes(5)}, \mathbf{primes(4)}, \mathbf{primes(3)}, \mathbf{primes(2)}\}, \{(R_2, 2)\}\rangle_3\\
&\langle\{primes(7)\sharp 1, primes(6)\sharp 2, primes(5)\sharp 3\}, \{\mathbf{primes(4)}, \mathbf{primes(3)}, \mathbf{primes(2)}\}, \{(R_2, 3)\}\rangle_4\\
&\langle\{primes(7)\sharp 1, primes(6)\sharp 2, primes(5)\sharp 3, primes(4)\sharp 4\}, \{\mathbf{primes(3)}, \mathbf{primes(2)}\}, \{(R_2, 4)\}\rangle_5\\
&\langle\{primes(7)\sharp 1, primes(6)\sharp 2, primes(5)\sharp 3, primes(4)\sharp 4, primes(3)\sharp 5\}, \{\mathbf{primes(2)}\}, \{(R_2, 5)\}\rangle_6\\
&\langle\{primes(7)\sharp 1, primes(5)\sharp 3, primes(4)\sharp 4, primes(3)\sharp 5\}, \{\mathbf{primes(2)}\}, \{(R_2, 5)\}\rangle_6\\
&\langle\{primes(7)\sharp 1, primes(5)\sharp 3, primes(4)\sharp 4, primes(3)\sharp 5, primes(2)\sharp 6\}, \{\}, \{\}\rangle_7\\
&\langle\{primes(7)\sharp 1, primes(5)\sharp 3, primes(3)\sharp 5, primes(2)\sharp 6\}, \{\}, \{\}\rangle_7
\end{aligned}$$

Note that we have left out state names as well as transitions due to space restrictions. The propagation store together with the constraint store remains constant when propagation occurs. When simplifying, their combined size decreases. It turns out that this is typical for terminating programs. □

A problem regarding the propagation store is that it can be infinitely large for some states in computations of a CHR program. To prevent this, we formulate conditions on propagation rules that guarantee that no sequence of propagation rules in computations of a CHR program can be infinitely long. Since such a condition is related to termination, we discuss it in the next section.

## 3 Termination of general CHR programs

Due to CHR's multi-headed rules and multiset semantics, it is often difficult to predict the behavior of CHR programs. Especially the concept of propagation, complicates reasoning about program properties, since it requires a fire-once policy on propagation rules to prevent trivial non-termination. New concepts are therefore required to handle propagation, which cannot be directly adapted from existing notions in LP an TRS termination analysis.

*Example 5.* Consider the following propositional CHR program,

$$R_1 \ @ \ a, a \Rightarrow b. \qquad\qquad R_2 \ @ \ b, b, b \Leftrightarrow a.$$

For a query with two constraints $a$, the program terminates as $R_1$ can only be applied in two different ways, as such only adding 2 constraints $b$. However, when querying with three $a$'s, the propagation rule can be applied in 6 different ways. As a consequence, the second rule becomes applicable twice, as such causing new tokens to be introduced. It is therefore a non-terminating program. □

Currently, two approaches exist, one for CHR without propagation and another for CHR with propagation. The first one, essentially adapts existing conditions in LP and TRS. It measures the size of the constraint store and guarantees decreases between consecutive CHR states. The second approach formulates conditions on adding constraints to the constraint store. In order to be able to handle propagation, it is strictly less powerful on CHR programs without propagation.

Although the approaches can trivially be combined into a global approach (by using one condition for simplification only programs, the other for programs with propagation), it would be satisfactory to formulate general conditions. In addition, this turns out to gain a new class of programs we can prove terminating.

*Example 6 (Problem class).* The program is a constructed example, representing a class of CHR programs which cannot be proved terminating using existing approaches. It contains both a propagation rule and a simplification rule and terminates for all finite queries.

$$R_1 \ @ \ a(s(N)), a(N), a(N) \Leftrightarrow a(s(N)), a(N). \qquad\qquad R_2 \ @ \ a(s(s(N))) \Rightarrow a(N).$$

As the program contains a propagation rule, only the condition for CHR with propagation can be used to prove termination. However, as the condition for CHR with propagation requires that $|a(s(s(N)))| > |a(N)|$ for the propagation rule, a decrease in the number of maximally ranked constraints in the simplification rule can never be shown. □

In this section, we present a general approach, capable of proving termination of an entirely new class of CHR programs of the kind as presented in the example above. Our conditions are strictly more powerful than those formulated in the existing approaches. We resolve therefore the drawbacks of existing approaches.

### 3.1 Preliminaries

First, we define some preliminary notions w.r.t. interpretations for constraints in CHR programs. These notions are adapted from the LP context [13–15]. Then, we introduce the existing approaches [10, 11].

$Term_P$ and $Con_P$ denote the sets of respectively all terms and all constraints that can be constructed from the alphabet underlying $P$. As in LP, we wish to describe the constraints that participate in computations of a CHR program $P$, given the constraints $S$ that are part of the query, by its call set. As such, we can establish interpretations which result in better approximations of the behavior of a CHR program.

**Definition 10 (Call set).** *Let $S \subseteq Con_P$. Then by $Call(P, S)$, we denote the subset of $Con_P$, such that $C \in Call(P, S)$ whenever $C$ is a constraint used to apply a rule in some computation of $P$ for $I$, where $I \subseteq S$.* □

To measure constraints, we use *norms* and *level mappings*. In general this is referred to as *interpretations* for CHR constraints. The sizes of consecutive computation states are compared using the *level values* (or *ranks*) of the constraints in the state. We recall their definition here in the context of CHR.

**Definition 11 (Norm, level mapping).** *Let $P$ be a CHR program. Then, a norm is a function $\|.\| : Term_P \to \mathbb{N}$ and a level mapping $|.| : Con_P \to \mathbb{N}$.* □

Several examples of norms and level mappings can be found in literature on LP termination analysis [13]. Two well-known norms are *list-length* and *term-size*. The most common kind of level mapping is the *linear level mapping*, where a constraint is mapped to a positive linear combination of the positive integer norms of its terms. We also require norms and level mappings to be *rigid* w.r.t. the constraints represented by the call set. That is, all constraints in the call set must have unique interpretations, which cannot alter under substitution.

***Termination of CHR programs without propagation.*** In [10], a condition for CHR programs without propagation is discussed. In such a setting, decreases are shown between consecutive CHR states. These are compared by using a multiset order [16] on the constraint store. If such a decrease is shown for every application of a rule, the program must terminate.

**Definition 12 (Multiset order).** *Let $r$ represent the level value of some atom and let $n_r^s$ represent the number of atoms of level $r$ in a multiset $s$. Then a multiset order is an induced order, given a level mapping $|.|$ for its atoms. A multiset $s$ is considered larger than a multiset $t$, denoted $s \succ_m t$, if some atom of level value $r$ exists, such that $n_r^s > n_r^t$ and such that $\forall q > r : n_q^s = n_q^t$.* □

The next example demonstrates how to prove termination of a CHR program with a multiset order on the constraint store.

*Example 7.* The program is terminating for all ground queries.

$R_1$ @ $a(s(N)), a(N), a(N) \Leftrightarrow a(s(N)), a(N)$. $\quad$ $R_2$ @ $a(s(N)) \Leftrightarrow a(N)$.

Termination is shown using a level mapping $|a(N)| = \|N\|$ and multiset order. □

***Termination of CHR programs with propagation.*** The ranking condition (RC) for CHR with propagation proves termination in an entirely different way [11]. It guarantees that only a finite number of constraints can be added to the constraint store. Because simplification removes constraints and propagation respects the fire-once policy, this implies termination.

The RC compares sizes of individual constraints, rather than multisets of constraints. It requires that propagation rules can only add constraints ranked strictly lower than any of the head constraints that gave cause to it. For simplification rules the number of constraints removed of maximal rank, has to be strictly greater than those added of maximal rank.

*Example 8.* The following example demonstrates the use of the RC for CHR with propagation.

$$R_1 \ @ \ a(s(s(N))) \Rightarrow a(s(N)), a(N). \qquad R_2 \ @ \ a(s(N)) \Leftrightarrow a(N).$$

Termination is shown for ground queries, using a level mapping from constraints $|a(N)| = \|N\|$ to the set of natural numbers. As such, a decrease in maximally ranked constraint exists in the second rule, while the first rule only adds constraints which are ranked strictly lower than those which fired the rule. □

Notice that the condition on simplification is a strengthened case of multiset order. Therefore, in the case of a program with only simplification rules, the condition for CHR without propagation covers more programs.

*Example 9.* We revisit Example 7.

$$R_1 \ @ \ a(s(N)), a(N), a(N) \Leftrightarrow a(s(N)). \qquad R_2 \ @ \ a(s(N)) \Leftrightarrow a(N).$$

Termination cannot be shown as the second rule requires that $|a(s(N))| > |a(N)|$. As such, no decrease in maximally ranked constraint can be shown for the first rule. The reason is that in the case of a propagation rule, e.g. $R_2 \ @ \ a(s(N)) \Rightarrow a(N)$, we obtain a non-terminating program. □

### 3.2 Termination of general CHR programs

As mentioned earlier, we can regard a computation in CHR as a subsequence of simplification steps, interleaved with sequences of propagation steps. A CHR program can only be guaranteed to terminate if these sequences of propagation steps cannot be infinitely long. The RC on propagation rules guarantees this.

**Definition 13 (RC on propagation rules).** *Let $R_p \ @ \ h_1, \ldots, h_n \Rightarrow G \mid b_1, \ldots, b_m$ be a propagation rule in a CHR program $P$, $I$ a query and $\sigma$ a match substitution for the heads of $R_p$ such that $CT \models \exists\theta : G\sigma\theta$ holds. Then, the RC on propagation rules is satisfied w.r.t. a rigid level mapping $|.|$ for $Call(P, I)$ iff $\forall h_i, b_j : |h_i\sigma\theta| > |b_j\sigma\theta|$.* □

At first sight, the condition on propagation seems unnecessarily strict. However, if we would not require that all body constraints are ranked strictly lower than any of the head constraints, it is possible that added constraints replace those which gave cause to them, enabling the rule to be fired at the same level of interpretation. We illustrate the RC by proving termination of the following example program without simplification rules.

*Example 10 (Fibonacci).* The program calculates Fibonacci numbers. The first rules resolve base cases, while the third rule adds Fibonacci constraints.

$R_1$ @ $fib(N, M) \Rightarrow N = 0 \mid M = 0.$
$R_2$ @ $fib(N, M) \Rightarrow N = s(0) \mid M = 1.$
$R_3$ @ $fib(s(s(N)), M_1), fib(s(N), M_2) \Rightarrow fib(N, M), M_1 \ is \ M_2 + M.$

As we will query the program with constraints of the type $fib(n, m)$ with $n$ ground and $m$ a variable, we measure these constraints by the term-size of their first argument. Therefore, $|fib(N, M)| = \|N\|_{ts}$ is rigid w.r.t. the call set. The first rules trivially satisfy the RC as no CHR constraints are added. As for the third rule the first argument decreases in term-size, we prove termination. □

The next proposition formulates that when the RC is satisfied for all propagation rules in a CHR program, there cannot exist infinite sequences of propagation steps in computations for the program.

**Proposition 1.** *If a CHR program $P$ with a query $I$ satisfies the RC on propagation rules w.r.t. a rigid level mapping $|.|$, then there cannot exist infinite propagation in a computation of $P$, if the constraint store $S$ is finite.*

*Proof.* We rank tokens according to the corresponding CHR constraint of smallest rank. Therefore, no token added by propagation can be of equal size or greater than the token removed. This implies that the multiset size of the token store decreases after every application of a propagation rule. No infinite sequences of propagation steps can therefore exist. □

When a CHR program satisfies the RC on propagation rules, then in any CHR state in a computation of the program, the propagation store is finite.

**Corrolary 1** *If a CHR program $P$ with a query $I$ satisfies the RC on propagation rules w.r.t. a level mapping $|.|$, then for any CHR state in a computation of $P$ for $I$, the propagation store $V$ is finite if the constraint store $S$ is finite.* □

We now come to the heart of our approach. As we could already observe in Example 4, when propagating, the combined size of constraint and propagation store $|chr(S) \uplus V|$ remains equal while the size of the token store $|T|$ decreases. We define therefore the following lexicographical interpretation for CHR states.

**Definition 14 (CHR state interpretation).** *The CHR state interpretation, is a lexicographical order of the following form:*

$$|\langle S, T\rangle_\nu| = \langle |chr(S) \uplus V|, |T|\rangle$$

*Here, $\uplus$ denotes multiset union, $chr(S) = \{chr(C) \mid C \in S\}$ the unlabeled variant of the constraint store, $V$ the propagation store and $T$ the token store. $chr(S)$ and $V$ are multisets of constraints. The tokens are measured by the smallest level value of a CHR constraint represented in the token: $|(R_i, id_1, \ldots, id_n)| = min\{|chr(c_1 \sharp id_1)|, \ldots, |chr(c_n \sharp id_n)|\}$.* □

We formulate a theorem stating that termination of a CHR program is guaranteed, when decreases are found in the CHR state interpretation, as defined above, for every application of a rule in a CHR program.

**Theorem 1.** *Let $P$ be a CHR program and $I$ a query. Furthermore, let $|.|$ be a rigid level mapping for the constraints in $Call(P, I)$. Then, a CHR program $P$ terminates for $I$ if for all applications of rules in $P$, using only constraints in $Call(P, I)$, the lexicographical ordering on CHR states decreases.*

*Proof.* The ordering we use is well-founded. Therefore, computations can only be finitely long. By definition, this implies termination of $P$. □

Note that whenever the RC on propagation rules is satisfied, we guarantee that there is a decrease in the CHR state interpretation for every application of a propagation rule. To observe decreases when simplifying on states, we must guarantee decreases in $|chr(S) \uplus V|$ as the token store may increase in size. Such decreases can be observed in Example 4 as well.

To formulate a condition on simplification rules, which guarantees decreases in $|chr(S) \uplus V|$, we first guarantee that a multiset decrease exists in the constraint store $chr(S)$. Such a decrease can be shown using the condition of Frühwirth [10].

**Definition 15 (RC on simplification rules).** *Let $R_s$ @ $h_{(s,1)}, \ldots, h_{(s,j_s)} \setminus h_{(s,j_s+1)}, \ldots, h_{(s,n_s)} \Leftrightarrow G_s \mid b_{(s,1)}, \ldots, b_{(s,m_s)}$ be a simplification rule in a CHR program $P$. Let $\sigma$ be a match substitution for the head constraints such that $CT \models \exists\theta : G_s\sigma\theta$ holds and let $|.|$ be a rigid level mapping w.r.t. a CHR program $P$ and a query $I$, such that the added and removed constraints in $R_s$ have ranks $r_1 > r_2 > ... > r_k$ and such that $n_i^a$ and $n_i^r$ represent respectively the number of constraints of rank $r_i$ added and removed by $R_s$. Then, $R_s$ satisfies the RC on simplification rules w.r.t. $|.|$ iff $\exists r_j : n_j^r > n_j^a$ and $\forall r_i > r_j : n_j^r = n_j^a$.* □

We already illustrated the RC in Example 7 on a program without propagation. However, when propagation is present, simplification rules cannot be considered separately. After all, by propagating on the added constraints of the simplification rule, the decrease caused in $chr(S)$ can still be undone. The constraints that can be added by propagation are represented in $V$. As a consequence, we have to refine our condition on simplification rules to guarantee that none of the constraints added to $V$ can undo the decrease caused in $chr(S)$. It is however impractical to analyze this multiset of added constraints.

Therefore, we observe the following. By definition, the tokens that enter the token store correspond to added constraints in the simplification rule. These

tokens can result in the addition of new constraints and thus new tokens. This process dies out when the RC on propagation rules is satisfied.

The application of a propagation rule is in one-to-one correspondence with a token. Given all tokens in the token store, we can apply their corresponding propagation rules simultaneously. The resulting multiset of added constraints from applying these rules simultaneously is referred to as the *layer 1 constraints* and is denoted by the multiset $L_1$. By adding these constraints, new tokens are added, resulting as such in layer two constraints $L_2$. For a finite propagation store, there are only a finite number of such layers: $V = L_1 \uplus L_2 \uplus \cdots \uplus L_v$.

Given the RC on propagation rules, we know that if constraints in $L_1$ cannot undo the multiset decrease in the constraint store that none of the subsequent layers $L_i$ can undo it either. This is because, if the added constraints in the first layer are already smaller than the relevant ranks in the simplification rule, than the added constraints in next layers need to be even smaller. Thus, they can definitely not influence the relevant ranks of the simplification rule. It is therefore sufficient to only consider $L_1$ constraints.

**Definition 16 (Refined RC on simplification rules).** *Consider a simplification rule $R_s$ @ $h_{(s,1)}, \ldots, h_{(s,j_s)} \setminus h_{(s,j_s+1)}, \ldots, h_{(s,n_s)} \Leftrightarrow G_s \mid b_{(s,1)}, \ldots, b_{(s,m_s)}$ in a CHR program $P$ that satisfies the RC on simplification rules for $I$ w.r.t. a rigid level mapping $|.|$. Therefore, $\exists r_j : n_j^r > n_j^a$ and $\forall r_i > r_j : n_j^r = n_j^a$. Then, the refined RC on simplification rules is satisfied for $R_s$ iff for all heads $h_{(p,i_p)}$ in propagation rules in $P$ for which $CT \models \exists \mu\sigma'\theta' : (G_s\sigma\theta \wedge (b_{(s,i_s)}\sigma\theta = h_{(p,i_p)}\mu) \wedge G_p\mu\sigma'\theta')$ holds: $r_j > |b_{(p,k)}\mu\sigma'\theta'|$ for all $b_{(p,k)}$. Here, $b_{(p,k)}$ is an added constraint in the propagation rule and $\mu$ a substitution for matching $b_{(s,i_s)}\sigma\theta$ with $h_{(p,i_p)}$. The substitutions $\sigma\theta$ come from matching and answer substitutions in the simplification rule and $\sigma'$ is a match substitution for the heads of the propagation rule and $\theta'$ a c.a.s. for satisfaction of the guard $G_p$ of the propagation rule.* □

We illustrate the refined RC by proving termination of the problem class from Example 6.

*Example 11 (Problem class).* The program terminates for ground queries.

$$R_1 \text{ @ } a(s(N)), a(N), a(N) \Leftrightarrow a(s(N)). \qquad R_2 \text{ @ } a(s(s(M))) \Rightarrow a(M).$$

By the RC on propagation rules, we require that $|a(s(s(M)))| > |a(M)|$. This is satisfied if we measure constraints by a term-size norm: $|a(N)| = \|N\|$. By the refined RC on simplification rules, we have that the constraints of rank $|a(N)|$ are decreased in number. By propagation on the added constraints this number cannot be increased again: $|a(M)| < |a(N)|$ when $a(s(N)) = a(s(s(M)))$. □

**Proposition 2.** *If a CHR program $P$ satisfies the refined RC for simplification rules w.r.t. a rigid level mapping $|.|$ for $Call(P, I)$, then there exists no infinite subsequence of simplification rules in a computation for $P$ with $I$.*

*Proof.* By the RC for simplification we know that $\exists r_j : n_j^r > n_j^a$ and $\forall r_i > r_j : n_j^r = n_j^a$. There is therefore a decrease in the size of the constraint store.

By considering the added constraints by propagating on the obtained state, we guarantee that the added constraints to the propagation store by simplification cannot undo this decrease. As such, there can only exist a finite subsequence of simplification steps in computations for $P$ with $I$. □

By Proposition 1 and 2, we guarantee decreases for every application of a rule in a CHR program if the RC on propagation rules and the refined RC on simplification rules are satisfied. Therefore, the CHR program must terminate.

**Theorem 2.** *A CHR program $P$ terminates for a query $I$ if it satisfies the RC for propagation and the refined RC for simplification w.r.t. a rigid level-mapping $|.|$ for $Call(P, I)$.*

*Proof.* By Proposition 1 and 2, a decrease in state size can be detected for each rule application. By Theorem 1, $P$ must terminate for $I$. □

## 4 Discussion

We evaluate our method first by comparing it to existing approaches. As stated earlier the condition on CHR without propagation requires a multiset decrease. In that setting, our method is as strong for the considered class of CHR programs.

**Proposition 3.** *The RC for CHR programs without propagation [10] is a special case of the RC for general CHR programs.*

W.r.t. the approach taken for CHR with propagation, our approach is more general as well. The condition formulated on propagation rules is identical to ours. However, our condition on simplification rules is more general. Our method also allows lower ranked constraints to be evaluated.

**Proposition 4.** *The RC for CHR programs with propagation [11] is a special case of the RC for general CHR programs.*

Therefore, our approach is able to prove strictly more programs terminating. w.r.t. both approaches. Consider the next example program.

*Example 12.*
$$R_1 \ @ \ list([D|L]), list(L) \Leftrightarrow list([D|L]).$$
$$R_2 \ @ \ list([A, B, C|L_1]), list([A, B, C|L_2]) \Rightarrow length(L_1, S_1), length(L_2, S_2), S_1 = S_2 \mid list(L_1), list(L_2).$$

Termination cannot be proved when using the existing approaches. Our approach can prove termination for queries with nil-terminated lists. As such the constraints in the call set can be measured by a rigid level mapping, list-length. In case of the propagation rule, the condition states that: $\forall h_i, b_j : |h_i\sigma\theta| > |b_j\sigma\theta|$.

$$|list([A, B, C|L_1])| > |list(L_1)| \qquad |list([A, B, C|L_2])| > |list(L_1)|$$
$$|list([A, B, C|L_1])| > |list(L_2)| \qquad |list([A, B, C|L_2])| > |list(L_2)|$$

For list-length these conditions are satisfied.

For the first rule, the number of constraints of rank $|list([D|L])|$ remains the same and the number of constraints of rank $|list(L)|$ is decreased in number. Now, we verify that none of the constraints added by propagating on the added constraint $list([D|L])$ can undo this decrease. That is,

$$|list(L)| > |list(L_1)| \text{ and } |list(L)| > |list(L_2)|$$

This is a consequence of matching the body of the simplification rule with the heads of the propagation rule: $[D|L] = [A, B, C|L_1]$ and $[D|L] = [A, B, C|L_2]$. Therefore, the number of constraints of maximally rank $|list([D|L])|$ remains the same, the number of constraint of rank $|list(L)|$ decreases. Both RCs are therefore satisfied w.r.t. list-length. The program therefore terminates for all queries of constraints with nil-terminated lists. □

Our approach is currently unable to handle the following kind of programs:

*Example 13.* The program terminates for ground queries.

$$R_1 \ @ \ a(s(N)), a(N), a(N) \Leftrightarrow a(s(N)). \qquad R_2 \ @ \ a(s(N)) \Rightarrow a(N).$$

Note that the program terminates because the simplification rule removes two constraints of rank $r_j$, while the propagation rule only adds one such constraint.

The refined RC on simplification rules cannot be satisfied for this program. The constraints of rank $|a(N)|$ are decreased in number in the first rule while the constraints added by propagating on the added constraints of the simplification rule are of the same rank. That is, $r_j = |b_{(p,k)}\mu\sigma'\theta'|$.

For single-headed propagation rules, we can however refine our condition by considering that no recombinations are required to fire these rule. Because of this direct correspondence, we do not have to assume that multiple instances of some constraint can be added. Therefore, for every added CHR constraint in rule 1, we add only one constraint by the propagation rule. The decrease caused in the constraint store is therefore not undone by propagation. In such cases, we can allow that $r_j = |b_{(p,k)}\mu\sigma'\theta'|$, given that the combined effect of multiple single-headed propagation rules does not undo the decrease. □

We will study this refinement in future work.

## 5 Conclusion

In this paper, we presented a new approach to termination analysis of CHR programs. Our approach proves termination by formulating conditions on the size of a CHR state. To measure states, we introduced a new interpretation, based on the constraint store, the token store and the propagation store. On the basis of these we compose a lexicographical interpretation and formulate conditions guaranteeing decreases between all consecutive CHR states. We showed that our approach generalizes the existing approaches and that it is able to prove termination of an entirely new class of CHR programs. In future work, we will refine the approach for single-headed propagation rules. We will develop an efficient termination tool to support thorough experimentation.

## References


1. Frühwirth, T.: Theory and practice of constraint handling rules. J. Log. Program. **37**(1-3) (1998) 95–138
2. Schrijvers, T.: Analyses, optimizations and extensions of constraint handling rules: Ph.d. summary. In: ICLP05. (2005) 435–436
3. Schrijvers, T., Frühwirth, T.: Optimal union-find in constraint handling rules. TPLP **6**(1-2) (2006) 213–224
4. Sneyers, J., Schrijvers, T., B., D.: The computational power and complexity of constraint handling rules. In: CHR05. (2005) 3–17
5. Duck, G., Stuckey, P., García de la Banda, M., Holzbaur, C.: The refined operational semantics of constraint handling rules. In: ICLP04. (2004) 90–104
6. Abdennadher, S., Marte, M.: University course timetabling using constraint handling rules. Applied Artificial Intelligence **14**(4) (2000) 311–325
7. Frühwirth, T., Brisset, P.: Optimal placement of base stations in wireless indoor telecommunication. In: CP98, Springer-Verlag (1998) 476–480
8. Duck, G., Stuckey, P., Sulzmann, M.: Observable confluence for constraint handling rules. In: CHR06. (2006) 61–76
9. Abdennadher, S., Frühwirth, T., Meuss, H.: On confluence of constraint handling rules. In: CP96, Springer-Verlag (1996)
10. Frühwirth, T.: Proving termination of constraint solver programs. In: New Trends in Constraints, Springer-Verlag (2000) 298–317
11. Voets, D., Pilozzi, P., De Schreye, D.: A new approach to termination analysis of constraint handling rules. In: CHR07. (2007)
12. Abdennadher, S.: Operational semantics and confluence of constraint propagation rules. In: CP97, Springer-Verlag (1997) 252–266
13. De Schreye, D., Decorte, S.: Termination of logic programs: the never-ending story. Journal of Logic Programming **19-20** (1994) 199–260
14. Bezem, M.: Characterizing termination of logic programs with level mappings. In: NACLP. (1989) 69–80
15. Dershowitz, N.: Termination of rewriting. Journal of Symbolic Computation **3**(1-2) (1987) 69–116
16. Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Commun. ACM **22**(8) (1979) 465–476

# Finally, A Comparison Between Constraint Handling Rules and Join-Calculus

Edmund S. L. Lam[1] and Martin Sulzmann[2]

[1] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
lamsoonl@comp.nus.edu.sg
[2] Programming, Logics and Semantics Group, IT University of Copenhagen
Rued Langgaards Vej 7, 2300 Copenhagen S Denmark
martin.sulzmann@gmail.com

**Abstract.** We provide a comparison between Constraint Handling Rules and Join-Calculus. Constraint Handling Rules is a concurrent constraint programming language originally designed for writing constraint solvers. Join-Calculus is a process calculus designed to provide the programmer with expressive concurrency abstraction. The semantics of both calculi is based on the Chemical Abstract Machine. Hence, we expect that both calculi share some commonalities. Surprisingly, both calculi have thus far been studied independently, yet we believe that a comparison of these two independent fields of study is long overdue. This paper establishes a first bridge between Constraint Handling Rules and Join-Calculus as a basis for future explorations. Specifically, we provide examples showing that Join-Calculus can benefit from guarded constraints and constraint propagation as found in Constraint Handling Rules. We provide a compilation scheme for such an enriched Join-Calculus by applying the constraint matching methods of the refined operational Constraint Handling Rules semantics.

## 1 Introduction

Constraint Handling Rules (CHR) [1] is a concurrent committed-choice constraint logic programming language to describe rewritings among multi-sets of constraints. Join-Calculus [2] is a process calculus designed to provide expressive concurrency abstractions in the form of reaction rules, known as Join-Patterns. Rule triggering depends on the availability and simultaneous consumption of messages received from various shared channels.

The Chemical Abstract Machine (CHAM) [3] provides the semantic foundations for both calculi. We therefore expect that both calculi share some common features. Surprisingly, CHR and Join-Calculus have been studied so far in complete isolation. We believe that a comparison between both calculi is long overdue and should enable a fruitful exchange of ideas and results. To the best of our knowledge, we are first to conduct such a comparison. In this paper, we make the following contributions:

Primitives:

| | | | |
|---|---|---|---|
| Process Name | $p$ | Variable | $x$ |
| Constant Value | $v$ | List of $a$'s | $\overline{a}$ |

Join-Calculus Essentials Expressions:

| | | | |
|---|---|---|---|
| Term | $t$ | ::= | $x \mid v$ |
| Process | $P$ | ::= | $p(\bar{t})$ |
| Concurrent Processes | $M$ | ::= | $P \mid M \ , \ M$ |
| Join-Pattern | $J$ | ::= | $P \mid J \mid J$ |
| Join-Body | $B$ | ::= | $P \mid B \mid B$ |
| Reaction Rule | $D$ | ::= | $J \rhd B$ |

**Fig. 1.** Join-Calculus Essentials

- We illustrate programming in CHR and Join-Calculus by example. It is known that both calculi are Turing-complete, hence, equally expressive. However, we show that Join-Calculus can benefit from having CHR style guarded and propagated constraints (Section 2.3).
- We introduce a new compilation scheme for Join-Patterns, which is essentially based on the CHR rule matching semantics (Section 3). This allows us to straight-forwardly introduce CHR features like guards, propagation and shared variables (non-linear patterns) into the Join-Pattern world.
- We investigate the commonalities and differences among the standard compilation schemes for rule matching of CHR rewrite rules and Join-Calculus reaction rules (Section 3.3).

Section 2 introduces Join-Calculus informally. We assume that the reader has some basic knowledge of CHR. We conclude and discuss future works in Section 4.

This paper is a revised and extended version of [4]. Our focus here is on a more detailed comparison among Join-Calculus and CHR. The issue of how to integrate CHR into a host language such as Haskell is left for future work.

## 2 Programming Examples

In this section, we informally introduce Join-Calculus via a simple example: a printer spooler coordinating a network of printers and clients which submits print jobs.

### 2.1 Join Calculus

Join-Calculus [2] is a process calculus that introduces an expressive high-level concurrency model, aimed at providing a simple and intuitive way to coordinate concurrent processes via reaction rules known as Join-Patterns.

Figure 1 shows the essential core Join-Calculus language. Processes are typically modeled as unique names $p$ each with a fixed number of term arguments. A

collection of concurrently running processes (denoted $M$) is represented by processes composed together with a binary operator ",". This collection is treated as an unordered set of processes. For instance, the following illustrates a collection of concurrent processes, representing the state of the printer spooler, denoted $\mathcal{S}$:

$$\mathcal{S} \quad = \quad \texttt{ready(p1)}, \texttt{ready(p2)}, \texttt{job(j1)}, \texttt{job(j2)}, \texttt{job(j3)}$$

A printer `Pm` which is available for printing will call the process `ready(Pm)`, while a print job `Jn` is submitted to the spooler via calling the process `job(Jn)`. We shall use standard CHR/Prolog notation to distinguish values and variables: Lowercase references for function/constant names and uppercase references for variables. Hence the above illustrates a state consisting of two available printers and three outstanding print jobs. A print job `Jn` is to be matched with any available printer `Pm`, during which printing can be initiated by sending `Jn` to `Pm` (`send(Pm,Jn)`). This behavior is captured by the reaction rule $\mathcal{D}$, defined as follows:

$$\mathcal{D} \quad = \quad \texttt{ready(Pm)} \mid \texttt{job(Jn)} \rhd \texttt{send(Pm,Jn)}$$

A reaction rule ($J \rhd B$) has two parts. We refer to the left-hand side $J$ as the Join-Pattern and to the right-hand side $B$ as the Join-Body (in our simplified setting rule processes). The Join-Pattern $J$ specifies that processes matching Join-Pattern $J$ can be consumed and replaced by rule processes $B$. Note that we will sometimes refer to the reaction rules as Join-Patterns as well if there is no ambiguity doing so. A set of reaction rules can be applied to a collection of concurrent processes. This is defined by two forms of transition steps, namely structural steps $(R \vdash M) \rightleftharpoons (R \vdash M')$ and reduction steps $(R \vdash M) \longrightarrow (R \vdash M')$ where $R$ is the set of reaction rules and $M$, $M'$ are collections of concurrent processes. This exploits the analogy that concurrent processes are a "chemical soup" of atoms and molecules, while reaction rules define chemical reactions in this chemical soup. Structural steps heat/cool atoms to and from molecules (switching to-and-fro ',' and '|'), while reduction steps apply reaction rules to the matching molecules. The following shows a possible sequence of structural/reduction steps which results from applying the printer spooler rule $\mathcal{D}$ on the spooler state $\mathcal{S}$:

$$\mathcal{D} = \texttt{ready(Pm)} \mid \texttt{job(Jn)} \rhd \texttt{send(Pm,Jn)}$$

$$\begin{array}{rl}
 & (\{\mathcal{D}\} \vdash \texttt{ready(p1)} \ , \ \texttt{ready(p2)} \ , \ \texttt{job(j1)} \ , \ \texttt{job(j2)} \ , \ \texttt{job(j3)}) \\
\rightleftharpoons & (\{\mathcal{D}\} \vdash \texttt{ready(p2)} \ , \ \texttt{job(j2)} \ , \ \texttt{job(j3)} \ , \ \texttt{ready(p1)} \mid \texttt{job(j1)}) \\
\longrightarrow & (\{\mathcal{D}\} \vdash \texttt{ready(p2)} \ , \ \texttt{job(j2)} \ , \ \texttt{job(j3)} \ , \ \texttt{send(p1,j1)}) \\
\rightleftharpoons & (\{\mathcal{D}\} \vdash \texttt{job(j3)} \ , \ \texttt{send(p1,j1)} \ , \ \texttt{ready(p2)} \mid \texttt{job(j2)}) \\
\longrightarrow & (\{\mathcal{D}\} \vdash \texttt{job(j3)} \ , \ \texttt{send(p1,j1)} \ , \ \texttt{send(p2,j2)})
\end{array}$$

When concurrent processes $J'$ matches a reaction rule $J \rhd B$ (ie. $J' = \theta(J)$ for some substitution $\theta$) causing the rule to be applied, we say that $J'$ has *triggered* the Join-Pattern $J$. Note the inherent non-determinism in matching processes with Join-Patterns: any pair of `ready(P)` and `job(J)` can be arbitrarily chosen by a structural step and matched with the Join-Pattern.

There are several implementations of Join-Calculus style concurrency abstractions. The JoCaml system [5] is one such example, which introduces Join-

Patterns into the programming language Caml. For instance, the printer spooler can be implemented in JoCaml as follows (omitting details of the function `send`):

```
let ready(P) & job(J) = send(P,J)
in ready(p1) & ready(p2) & job(j1) & job(j2) & job(j3)
```

As shown above, Join-Patterns in JoCaml are declared by the 'let' definition. There are some minor syntax differences ('$\triangleright$' reaction rule symbol is replaced by the more common '=') but it's intended meaning corresponds to it's Join-Calculus counterpart. Also, the symbol '&' replaces both '|' and ',' as a parallel composition operator for specifying both Join-Patterns and concurrent processes. The keyword '`or`' is used to concatenate multiple reaction rules in a 'let' definition. Process calls are treated just like ordinary procedural calls without return values, with the exception that they are matched with Join-Patterns.

On top of Join-Patterns, language extensions like JoCaml also introduces synchronous process calls which returns values, thus provide a synchronization mechanism among concurrent processes. We will omit this to focus our attention on Join-Calculus and CHR.

### 2.2 Constraint Handling Rules

Independently, Constraint Handling Rules (CHR) [1] has been developed in the field of constraint solving. CHR is a concurrent committed choice constraint programming language. Originally designed for writing constraint solvers, CHRs have through the years been exploited in a wide range of applications [6, 7]. A CHR program essentially consist of a set of multi-headed guarded rules, describing rewritings among multisets of constraints. These rewritings share a striking similarity with Join-Calculus and the chemical abstract machine reductions: CHR rules can be treated as the reaction rules and CHR constraint multiset as the chemical soup.

We illustrate this by reformulating the printer spooler reaction rule from the previous section, with the following CHR program and CHR derivation:

$$
\begin{array}{rcl}
\mathcal{P} & = & \{\texttt{print} \mathbin{@} \texttt{ready(P),job(J)} \Longleftrightarrow \texttt{send(P,J)}\} \\
 & & \\
 & & \{\texttt{ready(p1),ready(p2),job(j1),job(j2),job(j3)}\} \\
\rightarrowtail_{\mathcal{P}} & & \{\texttt{ready(p2),job(j2),job(j3),send(p1,j1)}\} \\
\rightarrowtail_{\mathcal{P}} & & \{\texttt{job(j3),send(p1,j1),send(p2,j2)}\}
\end{array}
$$

Concurrent processes are represented by the multiset constraint store and CHR derivation steps take the place of the chemical abstract machine reduction steps. Unlike the CHAM semantics which explicitly express matching via structural steps , CHR constraint matching is implicit within derivation steps.

CHR also supports other features like constraint propagation and rule guards which can be possible useful extensions to the Join-Pattern implementations. There are also several well-developed CHR operational semantics [8] and highly competitive state-of-the-art implementations [9, 10].

### 2.3 Extending Join-Patterns with Guards and Propagation

A particularly useful extension to the Join-Calculus language is Join-Patterns with guard constraints (also known as guarded Join-Patterns), which allows

the programmer to express boolean conditions on Join-Patterns. To illustrate guarded Join-Patterns, we consider a more complex variant of the printer spooler: Suppose that there are now conditions which must be met before a print job can be sent to a printer, namely:

- Print jobs have color requirements, namely black-and-white (`bw`), 16 bit color (`color16`) or 32 bit color (`color32`). Hence not all printers can execute a given print job.
- Only print jobs with authorized identity will be entertained.

To handle these requirements, we represent print jobs as `job(J,Cid,Cr)`, where additional arguments `Cid` and `Cr` are the client identifier and color requirements respectively. Available printers also additionally report their capabilities (ie. `ready(P,Cr)`). A new process `auth(Cid)` is also introduced to indicate that `Cid` is an authorized client. Note that we assume that the color values are ordered to reflect their increasing requirement levels (ie. `bw` $<$ `color16` $<$ `color32`). This new printer spooler can be implemented via the following guarded Join-Pattern, expressed in a pseudo JoCaml extension:

```
let auth(Cid1) & ready(P,Pcr) & job(J,Cid2,Jcr)
    when (Cid1 == Cid2) && (Pcr ≥ Jcr) = auth(Cid1) & send(P,J)
in ...
```

We assume that guard constraints are followed by the "`when`" keyword, and `&&` represents logical conjunction. Note that `auth(Cid1)` is "propagated", meaning that it is reintroduced in the Join-Body. This is because `Cid1` should remain authorized even after a print job is submitted.

Existing Join-Pattern implementations (eg. JoCaml [5], Polyphonic C# [11]) use a Join-Pattern compilation scheme [12] that maintains the states of Join-Patterns to determine when they can be triggered during runtime. (we briefly discuss this scheme in Section 3.1). Unfortunately, as indicated in [11], such compilations do not directly support Join-Patterns with guard constraints. It may seem that guarded Join-Patterns can be easily encoded in basic Join-Patterns. For instance, one attempt in JoCaml could be as follows:

```
let auth(Cid1) & ready(P,Pcr) & job(J,Cid2,Jcr) =
      if (Cid1 == Cid2) && (Pcr ≥ Jcr)
      then auth(Cid1) & send(P,J)
      else auth(Cid1) & ready(P,Pcr) & job(J,Cid2,Jcr)
in ...
```

This encoding uses a standard if-then-else conditional statement to replicate the semantics of a guarded Join-Pattern: if conditions are favourable then proceed as normal, otherwise abort by replenishing the involved processes for future matching. While this encoding ensure correctness (authorized print jobs are only sent to printers with sufficient capabilities), it cannot ensure completeness (all authorized print jobs will be submitted to any available and capable printer). This is because standard Join-Pattern compilation schemes do not test all combinations of concurrent processes, as it is unnecessary when matching is merely

a test of presence/absence of processes (This is true for standard Join-Patterns). To date, no existing implementations provide efficient and practical compilation of Join-Patterns with guard constraints.

Yet guard constraints are natively supported in the CHR framework. Furthermore, CHR provides other features like propagation and non-linear patterns (variables appearing in multiple unique locations), which would be obviously useful if available in Join-Patterns. For instance, we can represent the new printer spooler Join-Pattern as the following CHR rule:

$$\texttt{print} \ @ \ \texttt{auth(Cid)} \setminus \texttt{ready(P,Pcr),job(J,Cid,Jcr)} \iff \texttt{Pcr} \geq \texttt{Jcr} \mid \texttt{send(P,J)}$$

CHR rules natively support propagation: constraints like `auth(Cid)` are known as propagated head (appearing before the '\' symbol), which are necessary for triggering the rule but not deleted after its application. Variables are also allowed to appear in multiple locations of the rule head (non-linear patterns). This approach will also benefit from well-developed CHR operational semantics [8] as well as existing CHR optimizations such as hash indexing and optimal join-ordering. Hence we believe that the CHRs would be the ideal solution to handle guarded Join-Patterns, offering a highly optimized and well-studied multiset rewriting operational semantics that shares a similar foundational semantics (chemical abstract machine).

## 3 Compilation Schemes

In this section, we review the compilation scheme which is currently the de-facto standard for Join-Pattern implementations. Following this, we introduce a new compilation scheme which is based on CHR rule matching.

### 3.1 Standard Join-Pattern Compilation Scheme

Existing Join-Pattern implementations compile Join-Patterns into state machines that maintain the matching states of the Join-Patterns [12]. For instance, in JoCaml, this compilation involves constructing $n$ message channels (which are typically queues) and a finite state machine (automaton) for each '`let`' definition, that keeps track of the *matching status* of the $n$ queues. Note that '`let`' definitions can contain any finite number of Join-Patterns delimited by the '`and`' keyword. Each message channel is assigned to a unique process name, and represents the collection of calls to this process by concurrent computation threads. Hence, a call to a process is analogous to the arrival of a new message in the corresponding message channel.

States of this automaton are essentially tuples of $n$ bits, one assigned to each message queue stating whether it is empty (`0`) or non-empty (`N`). This automaton is updated every time a new process is call (ie, a new message has arrived) or when a Join-Pattern is successfully matched. Figure 2 shows an example of a `let` definition, consisting of two Join-Patterns, as well as it's corresponding matching status automaton that is constructed. We label the first Join-Pattern as `J1` and the other as `J2` (Note that this labeling is for our presentation only and not part of the semantics).

Each edge labeled with a transition label, which is either of the form `m-j` stating that arrival of message `m` has triggered Join-Pattern `j`, or just `m` which states

**Implementation in JoCaml:**

```
let p() & q() = a()     (J1)
or p() & r() = b()      (J2)
in ...
```

**Chemical Abstract Machine Interpretation:**

$(\{J1, J2\} \vdash ...)$
where $J1 = p() \mid q() \rhd a()$
$J2 = p() \mid r() \rhd b()$

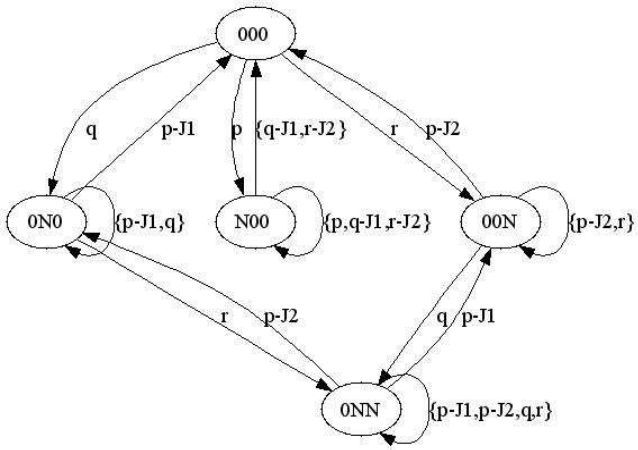



**Fig. 2.** A Matching Status Automaton for two Join-Patterns

that arrival of message `m` has triggered nothing and therefore is just queued. If there are more than one alternative transitions between two states of the automaton, we will represent them as a single edge with the set of alternative transitions. Since the Join-Patterns considered do not have guard conditions, messages channels are normally implemented with shared queue data structures, and the task of triggering Join-Patterns is as simple as popping messages from the relevant sets of queues. The choice of taking `p-J1` to `000` or `0N0` depends on whether there are any more `q`'s left after `J1` is triggered.

A state automaton compilation like this implements a matching policy referred to as "match as soon as possible", where join patterns are immediately triggered upon the arrival of the final message that complete it's match. For example, suppose that all queues are currently empty (`000`), the arrival of `q` transits the automaton to `0N0` and does not triggering any join patterns since no complete matches are available. The arrival of `p` completes `J1`'s match and triggers it, hence the automaton makes the `p-J1` transition. Note that the "match as soon as possible" matching policy is sound with respect to the chemical abstract machine semantics, but does not allow all its possible non-deterministic behaviors. We will discuss more of this issue in Section 3.3.

### 3.2 CHR Rule Matching Compilation Scheme

We introduce a new Join-Pattern compilation scheme, based on CHR rule matching semantics. The idea is to compile Join-Patterns into CHR rules, where known CHR operational semantics can be applied to derive Join-Pattern triggering. Our

**CHR Primitives:**

| | | | |
|---|---|---|---|
| Constraint Names | $c$ | Variables | $x$ |
| Constant Values | $v$ | List of $a$'s | $\overline{a}$ |

**CHR Terms and Constraints:**

| | | | |
|---|---|---|---|
| Substitutions | $\theta$ | ::= | $[v_1/x_1, .., v_n/x_n]$ |
| Terms | $t$ | ::= | $x \mid v$ |
| CHR Constraints | $C$ | ::= | $c(\bar{t})$ |
| Numbered Constraints | $Cn$ | ::= | $C\#n$ |
| Occurrence Constraints | $Co$ | ::= | $C : i$ |
| Active Constraints | $A$ | ::= | $C\#n : i$ |

**CHR Matching Sets:**

| | | | | | |
|---|---|---|---|---|---|
| Rule Head | $H$ | ::= | $Co$ | $\mid$ | $H \wedge H$ |
| Matching Set | $\mathcal{P}$ | ::= | $\{H\}$ | $\mid$ | $P \uplus P$ |

**CHR Stores and States:**

| | | | | |
|---|---|---|---|---|
| Stores | $St$ | ::= | $\emptyset \mid \{Cn\} \uplus St$ | |
| Match States | $\sigma$ | ::= | $\langle C, St\rangle_n$ | (Initial) |
| | | $\mid$ | $\langle A, St\rangle_n$ | (Intermediate) |
| | | $\mid$ | $\langle \theta, R, St\rangle_n$ | (Match Success) |
| | | $\mid$ | $\langle \epsilon, St\rangle_n$ | (Match Fail) |

**Fig. 3.** CHR Rule Matching Essentials

presentation here is inspired by the refined CHR operational semantics [8]. Figure 3 reviews the essential components of the CHR language. The actual CHR framework is much richer than presented here (eg. guards, propagation, etc..). We will omit these features for simplicity, but note that extending this scheme with guards and propagation is straight-forward.

Lists are denoted by $[x \mid xs]$, where $x$ is the first element and $xs$ the tail. The empty list is denoted by $[\,]$ and $\bar{x}$ is short for a list of $x$'s. Sets are denoted by $\{x_1, ..., x_n\}$ and multi-set union of two sets $S_1$ and $S_2$ is denoted by $S_1 \uplus S_2$.

We are particularly interested in the multi-set matching part of CHR executions, hence we only consider CHR rule heads (rule bodies are omitted). Constraints in rule heads are assigned unique occurrence numbers (eg. $C : i$) with respect to there textural order in the program. Rule heads are matched against constraints in the multi-set store. Stored constraints are numbered (eg. $C\#n$) to distinguish duplicate copies. Constraint matching is driven by an active constraint, $C\#n : i$, which matches numbered constraint $C\#n$ with occurrence $i$. A matching set $\mathcal{P}$ is a set of CHR rule heads. We define two auxiliary functions $cons$ and $maxOccurs$, which returns constraints from numbered constraints and returns the maximum occurrence number from a matching set respectively.

Single-Step Matching Reduction: $\sigma \rightarrow_{\mathcal{P}} \sigma$

(Activate) $$\langle C, St\rangle_n \rightarrow_{\mathcal{P}} \langle C\#n : 1, \{C\#n\} \uplus St\rangle_{n+1}$$

(Match) $$\frac{\begin{array}{c} H_1' \wedge C' : j \wedge H_2' \in \mathcal{P} \\ \exists\theta \text{ such that} \quad \theta(C') = C \quad \theta(H_1') = cons(H_1) \quad \theta(H_2') = cons(H_2) \end{array}}{\langle C\#m : j, \{C\#m\} \uplus H_1 \uplus H_2 \uplus St\rangle_n \rightarrow_{\mathcal{P}} \langle \theta, H_1' \wedge C' : j \wedge H_2', St\rangle_n}$$

(Continue) $$\frac{j < maxOccur(\mathcal{P})}{\langle C\#m : j, St\rangle_n \rightarrow_{\mathcal{P}} \langle C\#m : (j+1), St\rangle_n}$$

(Deactivate) $$\frac{j \geq maxOccur(\mathcal{P})}{\langle C\#m : j, St\rangle_n \rightarrow_{\mathcal{P}} \langle \epsilon, St\rangle_n}$$

Exhaustive Matching Reduction: $\sigma \rightarrow_{\mathcal{P}}^* \sigma$

(Transitive) $$\frac{\sigma \rightarrow_{\mathcal{P}} \sigma' \quad \sigma' \rightarrow_{\mathcal{P}}^* \sigma''}{\sigma \rightarrow_{\mathcal{P}}^* \sigma''}$$

(Match-Success) $$\frac{\sigma \rightarrow_{\mathcal{P}} \langle \theta, R, St\rangle_n}{\sigma \rightarrow_{\mathcal{P}}^* \langle \theta, R, St\rangle_n}$$

(Match-Fail) $$\frac{\sigma \rightarrow_{\mathcal{P}} \langle \epsilon, St\rangle_n}{\sigma \rightarrow_{\mathcal{P}}^* \langle \epsilon, St\rangle_n}$$

**Fig. 4.** CHR Multi-set Rule Matching Semantics

Figure 4 formally specifies the CHR rule matching semantics. This semantics is defined in terms of reduction steps ($\rightarrow_{\mathcal{P}}$) which maps matching states to matching states. Matching starts from an initial matching state $\langle C, St\rangle_n$. Transition rules are tried in sequence. Rule (Activate) begins the matching procedure by activating constraint $C$. This involves adding $C$ to the store and assigning it occurrence number 1. The rule (Match) specifies the successful matching of a CHR rule. Active constraint $C\#m : i$ matches with the $i^{th}$ occurrence of $\mathcal{P}$ and matching partners $H_1$ and $H_2$ are present in the store. This leads to the match state $\langle \theta, R, St\rangle_n$ where $\theta$ is the matching substitution, $R$ is the rule heads that is matched and $St$ is the remaining store after matching constraints are removed. Rule (Continue) moves the search forward by incrementing the occurrence number of the active constraint, while (Deactivate) ends the search when last occurrence is tried and no match is found. Exhaustive reductions $\rightarrow_{\mathcal{P}}$ defines the reduction sequence from a initial match state to a final state (match success/match fail).

Figure 5 illustrates the CHR-Based Join-Pattern matching semantics. We assume a straight forward compilation scheme for Join-Patterns. Namely, processes (messages) are treated as constraints and Join-Patterns are assigned occurrence numbers, depending on the textual order of their appearance. A CHR-Based Join-Pattern matching state $(R \vdash M, St, n)$ consist of the reaction rules $R$, the concurrent messages/processes $M$ (ie. chemical soup), a CHR store $St$ and store identifier $n$. It essentially maintains the matching status of a set of

**Primitives:**

| | | | |
|---|---|---|---|
| Process Name | $p$ | Variable | $x$ |
| Constant Value | $v$ | List of $a$'s | $\overline{a}$ |

**Compiled Join-Pattern Expressions:**

$$
\begin{array}{rrcl}
\text{Term} & t & ::= & x \mid v \\
\text{Process/Constraint} & P & ::= & p(\bar{t}) \\
\text{Concurrent Processes} & M & ::= & P \mid M\ ,\ M \\
\text{Compiled Join-Pattern} & J & ::= & P : i \mid J \wedge J \\
\text{Reaction Rules} & R & ::= & \{J \rhd M\} \mid R \uplus R
\end{array}
$$

**CHR-Based Join-Pattern Matching State:**

$$
\begin{array}{rrcll}
\text{Matching State} & \mathcal{E} & ::= & (R \vdash M, \emptyset, 1) & \text{(Initial)} \\
 & & \mid & (R \vdash \emptyset, St, n) & \text{(Final)} \\
 & & \mid & (R \vdash M, St, n) & \text{(Intermediate)}
\end{array}
$$

**CHR-Based Reduction Step:** $(R \vdash M, St, n) \longrightarrow (R \vdash M, St, n)$

$$
\text{(Join-Pattern Triggered)}\quad \frac{R = \{H_1 \rhd B_1, ..., H_n \rhd B_n\} \quad \mathcal{P} = \{H_1, ..., H_n\} \quad \langle p(\bar{t}), St\rangle_n \rightarrow^*_{\mathcal{P}} \langle \theta, H_i, St'\rangle_{n'} \quad \text{for some } i \in \{1, ..., n\}}{(R \vdash \{p(\bar{t})\} \uplus M, St, n) \longrightarrow (R \vdash \theta(B_i) \uplus M, St', n')}
$$

$$
\text{(Message Stored)}\quad \frac{R = \{H_1 \rhd B_1, ..., H_n \rhd B_n\} \quad \mathcal{P} = \{H_1, ..., H_n\} \quad \langle p(\bar{t}), St\rangle_n \rightarrow^*_{\mathcal{P}} \langle \epsilon, St'\rangle_{n'}}{(R \vdash \{p(\bar{t})\} \uplus M, St, n) \longrightarrow (R \vdash M, St', n')}
$$

**Fig. 5.** CHR-Based Join-Pattern Matching Semantics

Join-Pattern reaction rules. Hence, it is the runtime structure that replaces the matching state automaton of standard Join-Pattern compilations. For instance, the following shows Join-Patterns (in JoCaml syntax) and its CHR-Based interpretation:

| **Implementation in JoCaml:** | | **CHR-Based Matching State:** |
|---|---|---|
| `let p() & q() = a()` | `(J1)` | $(\{$`J1,J2`$\} \vdash ...)$ |
| `or p() & r() = b()` | `(J2)` | where `J1` $=$ `p():1` $\wedge$ `q():2` $\rhd$ `a()` |
| `in ...` | | `J2` $=$ `p():3` $\wedge$ `r():4` $\rhd$ `b()` |

Reduction steps $\longrightarrow$ are defined by two transitions (Join-Pattern Triggered) and (Message Stored), both of which specify arbitrary scheduling of a process for CHR matching (Figure 4) and the respective outcomes of the matching. Transition (Join-Pattern Triggered) is taken when scheduled process is successfully matched (according to CHR matching semantics), hence the corresponding

Join-Pattern is triggered. Transition (Message Stored) is taken when scheduled process fails to match, hence the process is stored as a "message" in the CHR store, awaiting for future matching.

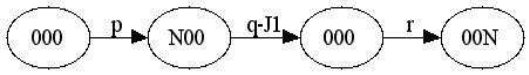
### 3.3 Discussion

**Non-Determinism in CHR Matching Semantics:** One important characteristic of this compilation scheme is that multiple CHR matchings are not intended to be done in parallel, but in an interleaving manner (transitions (Join-Pattern Triggered) and (Message Stored) are performed as "atomic" steps, hence cannot be intermediately interleaved). While this would likely limit performance of an actual implementation, it simplifies our CHR matching semantics by a great deal by not allowing execution of multiple active constraints in parallel, thus not introducing more non-determinism than needed. Note that our CHR matching semantics is no more non-deterministic than the refined CHR operational semantics $\omega_r$ [8] (Given a fixed sequence of goal constraints to activate, our matching semantics will derive similar results). The main additional source of non-determinism is instead introduced by the top level Join-Pattern reduction steps, which allows processes (constraints) to be scheduled for activation in arbitrary sequences.

**"Match as soon as Possible" versus CHR Matching Semantics:** The CHR matching semantics shares remarkable similarities with the Join-Pattern state automata "match as soon as possible" matching policy. The Join-Pattern state automata trigger Join-Patterns immediately once a complete match has "arrived", by keeping track on which message channels are empty/non-empty. Similarly, CHR matching semantics executes constraints in sequence of activation and triggers a rule immediately when the CHR store consists of a complete rule head match. Let's consider an example by examining the state transitions taken by the state automaton of Figure 2 in response to the message sequence, `p()`, `q()` then `r()`:

**Message Sequence:** `[p(),q(),r()]`

**Join-Pattern Trigger via State Automata (Figure 2):**




This results in the triggering of the join pattern `J1` as it is triggered immediately from the state `N00` once `q()` arrives. This finally leads to a state where only `r()` is left (`00N`). We consider the CHR matching of this example. For brevity, we omit the top-level Join-Pattern reduction steps, but we illustrate the underlying CHR derivations in the sequence of activation: `[p(),q(),r()]`

**Join-Pattern Trigger via CHR Matching (Figure 4 and 5):**

**CHR rule head patterns:**
$\mathcal{P} = \{\texttt{J1},\texttt{J2}\}$ where $\texttt{J1} = \texttt{p():1} \wedge \texttt{q():2}$ , $\texttt{J2} = \texttt{p():3} \wedge \texttt{r():4}$

**Activation Sequence:** `[p(),q(),r()]`

$$
\begin{array}{ll}
 & \langle \texttt{p()}, \emptyset \rangle_1 \\
\text{(Activate)} & \rightarrow_{\mathcal{P}} \langle \texttt{p()\#1} : 1, \{\texttt{p()\#1}\} \rangle_2 \\
\text{(Continue)} \times 4 & \rightarrow_{\mathcal{P}} \langle \texttt{p()\#1} : 5, \{\texttt{p()\#1}\} \rangle_2 \\
\text{(Deactivate)} & \rightarrow_{\mathcal{P}} \langle \epsilon, \{\texttt{p()\#1}\} \rangle_2
\end{array}
$$

$$
\begin{array}{ll}
 & \langle \texttt{q()}, \{\texttt{p()\#1}\} \rangle_2 \\
\text{(Activate)} & \rightarrow_{\mathcal{P}} \langle \texttt{q()\#2} : 1, \{\texttt{p()\#1}, \texttt{q()\#2}\} \rangle_3 \\
\text{(Continue)} & \rightarrow_{\mathcal{P}} \langle \texttt{q()\#2} : 2, \{\texttt{p()\#1}, \texttt{q()\#2}\} \rangle_3 \\
\text{(Match)} & \rightarrow_{\mathcal{P}} \langle [\,], \texttt{J1}, \emptyset \rangle_3
\end{array}
$$

$$
\begin{array}{ll}
 & \langle \texttt{r()}, \emptyset \rangle_3 \\
\text{(Activate)} & \rightarrow_{\mathcal{P}} \langle \texttt{r()\#1} : 1, \{\texttt{r()\#1}\} \rangle_4 \\
\text{(Continue)} \times 4 & \rightarrow_{\mathcal{P}} \langle \texttt{r()\#1} : 5, \{\texttt{r()\#1}\} \rangle_4 \\
\text{(Deactivate)} & \rightarrow_{\mathcal{P}} \langle \epsilon, \{\texttt{r()\#1}\} \rangle_4
\end{array}
$$

Similarly, the CHR derivation shows the triggering of rule `J1` and the final CHR store corresponding to `00N` in the state automaton. In both case, Join-Pattern (CHR rule) `J1` is triggered as the match {`p(),q()`} is completed first.

**Deleting Matching Transition versus Rule Ordering:** CHR rule heads and Join-Patterns may contain overlapping partial matches. Such overlaps are sources of non-deterministic behaviors in the theoretical CHR semantics and Join-Calculus semantics based on the chemical abstract machine. Overlapping partial matches allow multiple rules (CHR rule / reaction rule) to be applicable from certain states, and applying different rules in such states may lead to different outcomes. For example, given Join-Patterns in Figure 2, suppose we are in a state with messages `q()` and `r()`, according to chemical abstract machine, we can trigger either `J1` or `J2` is a `p()` arrives next. This situation is captured by the state `0NN` of the automaton in Figure 2 where we have two edges (`p-J1` and `p-J2`), both triggered by the message `p()`. This indicates that we can trigger either of the two Join-Patterns. In standard Join-Pattern implementations (eg. JoCaml), this is dealt with by allowing the compiler to choose an arbitrary transition edge and remove the other (compiler deletes either `p-J1` or `p-J2`). Hence the final matching status automaton generated would not exhibit such non-determinism.

For the CHR matching semantics, overlapping rules are dealt with in a less ad-hoc way. Consider the CHR matching reductions of this scenario (we assume that CHR store already contains constraints `q()` and `r()` when `p()` is activated:

$$
\begin{array}{ll}
 & \langle \texttt{p()}, \{\texttt{q()\#1}, \texttt{r()\#2}\} \rangle_3 \\
\text{(Activate)} & \rightarrow_{\mathcal{P}} \langle \texttt{p()\#3} : 1, \{\texttt{q()\#1}, \texttt{r()\#2}, \texttt{p(X)\#3}\} \rangle_4 \\
\text{(Match)} & \rightarrow_{\mathcal{P}} \langle [\,], \texttt{J1}, \{\texttt{r()\#2}\} \rangle_4
\end{array}
$$

Note that the active constraint $\texttt{p()\#3} : j$ can only match with occurrence $j$ of the CHR program, hence $\texttt{p()\#3} : 1$ will always try matching with `J1` first before taking transitions (Continue) twice to reach $\texttt{p()\#3} : 3$ which will try matching with `J2` (Note this never happen since matching with `J1` succeeds). Thus rule textual ordering would prevent such non-deterministic behaviors.

In essence, the CHR refined operational semantics is comparable to a Join-Pattern matching status automaton with a "match as soon as possible" policy and overlapping transitions are deleted depending on textual ordering (ie. transitions involving lower textual ordering join-patterns are deleted). Interestingly, researchers of the two communities proposed very different motivations for enforcing determinism:

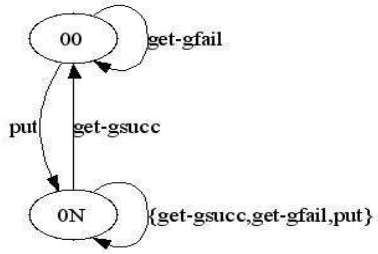
- **Join-Pattern matching status automata:** Non-determinism is removed purely for efficiency (we don't need to decide which overlapping rules to trigger at runtime, because there is at most one), doing so have a price in terms of semantics [12] as some behaviors stated by the theoretical calculus (ie. the chemical abstract machine) can no longer be observable. This suggests that deletion of non-deterministic transitions is viewed as a trade off favouring efficiency over modeling non-deterministic behaviors of concurrent programs.
- **CHR refined operational semantics:** Determinism is motivated not only because of efficiency, but also to make rule-based constraint programming easier (determinism enhances properties like termination and confluence [8]) Rule ordered executions also allow us to write more deterministic programs, which would not work as intended in a unordered setting.

Rule ordered matching semantics introduced by CHR would allow us to write more expressive Join-Patterns. This is illustrated by the following:

```
let get(X) & put(Y) = got(X,Y)      (gsucc)
or get(X) = got(X,ϵ)                (gfail)
in ...
```

The intend is to model a shared buffer, where `get` retrieves an object if one exists, retrieves nothing otherwise, while `put` places an object in the buffer. We assume that $\epsilon$ denotes the null object. According to the CHR matching semantics, an active `get(X)` will always try to match in rule order (ie. try `gsucc` before `gfail`). Hence this effectively models the firing of `gfail` in the absence of `put(y)` (ie. get nothing if there are no `put(Y)` found).

The matching status automaton generated by the standard Join-Pattern compilation scheme is illustrated by the following:




Note that arrival of a `get` message at state `0N` will cause two possible transitions (`get-gsucc` or `get-gfail`). Hence, according to standard compilation

practice, the compiler will choose either of the two transitions of state `ON` to be deleted from the final automaton. If the `get-gsucc` transitions are removed, only `gfail` will ever be triggered at runtime. If `get-gfail` is removed instead we get the desired behavior for this example. Unfortunately, this choice is not observable to the programmer, hence we cannot make any assumptions on ordering of Join-Patterns in standard Join-Pattern compilations, unlike in our CHR compilation.

**Performance versus Expressiveness:** Competitive implementations of Join-Patterns (eg. Polyphonic C# [11]) do not explicitly construct state automata but represent Join-Pattern matching states as bitmaps. Hence triggering of Join-Patterns can be executed in constant time, with known bit-masking techniques. The main disadvantage of using such compilation scheme is its incompatibility with the introduction of guard conditions. Our CHR compilation scheme benefits from the straight-forward extensions of features like propagation, guards and non-linear patterns. We also benefit from existing CHR optimizations (eg. constraint indexing, optimal join ordering, passive occurrences, etc.. [8, 13, 14] ). Most of such optimizations however, benefit only programs which uses guard conditions and non-linear patterns. Thus, for the class of programs which use only basic Join-Patterns, it is likely that our CHR matching compilation scheme will perform less efficiently compared to the standard schemes.

## 4 Conclusion and Future Work

We introduced a new Join-Pattern compilation scheme, based on the CHR matching semantics. This matching semantics is inspired by the CHR refined operational semantics. We have shown the basic difference and similarity between the standard Join-Pattern compilation scheme and CHR matching compilation scheme. The main benefits of our CHR matching compilation scheme is the possibility of extension with CHR features like guards and propagation, which will prove to be extremely useful.

An extension of Join-Patterns which introduces algebraic pattern matching in the matching of Join-Patterns is studied in [15]. Our approach generalizes this, as CHR matching semantics handles pattern matching over constraint (message) variables.

In the future, we intend to explore this relation further by implementing a prototype Join-Pattern system based on the CHR matching compilation scheme. Our CHR matching compilation scheme here is inherently single threaded, yet practical implementations would demand a system which is capable of executing matchings in parallel. Our works in a parallel implementation of CHR [16] would provide the framework for a prototype system based on parallel CHR matching. We also intend to investigate the implications of such implementations on performance, as well as on theoretical CHR properties (confluence, determinism).

## Acknowledgments

We thank Simon Peyton Jones for suggesting to establish formal links between Join-Pattern and CHR matching compilation schemes. We thank Matthias Zenger

for some initial discussions on how to encode Join-Calculus in Constraint Handling Rules. We thank the reviewers of CHR'08 for their useful comments on how to improve the paper.

## References

1. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37**(1-3) (1998) 95–138
2. Fournet, C., Gonthier, G.: The reflexive cham and the join-calculus. In: POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (1996) 372–385
3. Berry, G., Boudol, G.: The chemical abstract machine. In: POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (1990) 81–94
4. Sulzmann, M., Lam, E.S.L.: Haskell – Join – Rules. In Draft Proc. of IFL'07 (September 2007)
5. Conchon, S., Fessant, F.L.: Jocaml: Mobile agents for objective-caml. In: ASAMA '99: Proceedings of the First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents, Washington, DC, USA, IEEE Computer Society (1999) 22
6. P.J.Stuckey, Sulzmann, M.: A systematic approach in type system design based on constraint handling rules. Technical report (2001)
7. Alberti, M., Gavanelli, M., Lamma, E., Chesani, F., Mello, P., Torroni, P.: Compliance verification of agent interaction: A logic-based software tool. Applied Artificial Intelligence **20**(2-4) (April 2006) 133–157
8. Duck, G.J.: Compilation of Constraint Handling Rules. PhD thesis, The University of Melbourne (2005)
9. Schrijvers, T., Demoen, B.: The K.U.Leuven CHR system: implementation and application. In Frühwirth, T., Meister, M., eds.: First Workshop on Constraint Handling Rules: Selected Contributions. (2004) ISSN 0939-5091.
10. Van Weert, P., Schrijvers, T., Demoen, B.: K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In: 2nd Workshop on Constraint Handling Rules, Sitges, Spain. (2005) 47–62
11. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for c#. ACM Trans. Program. Lang. Syst. **26**(5) (2004) 769–804
12. Fessant, F.L., Maranget, L.: Compiling join-patterns. In: HLCL '98: High-Level Concurrent Languages, volume 16(3) of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, Sept. 1998. (1998)
13. Holzbaur, C., de la Banda, M.J.G., Stuckey, P.J., Duck, G.J.: Optimizing compilation of Constraint Handling Rules in HAL. TPLP **5**(4-5) (2005) 503–531
14. Schrijvers, T.: Analyses, optimizations and extensions of Constraint Handling Rules: Ph.D. summary. In: Proc. of ICLP'05. Volume 3668 of LNCS., Springer-Verlag (2005) 435–436
15. Ma, Q., Maranget, L.: Algebraic pattern matching in join calculus. LMCS-4 **1** (2008) 7
16. Sulzmann, M., Lam, E.S.L.: Parallel execution of multi set constraint rewrite rules. In proc. 10th International Symposium on Principles and Practice of Declarative Programming (July 2008)

# Verification of Constraint Handling Rules using Linear Logic Phase Semantics

Rémy Haemmerlé[1] and Hariolf Betz[2]

[1] CLIP Group, Facultad de Informática, Universidad Politécnica de Madrid, Spain
remy@clip.dia.fi.upm.es
[2] Faculty of Engineering and Computer Sciences, University of Ulm, Germany
hariolf.betz@uni-ulm.de

**Abstract.** Constraint Handling Rules (CHR) is a declarative concurrent programming language. Like the class of Concurrent Constraint (CC) languages, CHR features a declarative semantics based on Girard's intuitionistic linear logic. The phase semantics of linear logic has been used in the past to prove safety properties for the class of CC languages. In this paper we show that we can adapt this result to prove safety properties for CHR as well.

## 1 Introduction

Constraint Handling Rules (CHR) is a concurrent committed-choice rule-based programming language introduced in the 1990s by Frühwirth [1]. While it has been originally designed for the design and implementation of constraint solvers, it has come into use as a general-purpose concurrent programming language. Owing to its origins in the tradition of logic and constraint logic programming, CHR features a classical declarative semantics. Recently, Betz and Frühwirth [2, 3] proposed an alternative declarative semantics based on Girard's Linear Logic (LL) [4].

The class of Concurrent Constraint programming language (CC) was introduced by Saraswat in 1987 [5] as an unifying framework for constraint logic programming and concurrent logic programming with a synchronisation mechanism bases on constraint entailment. From the logic programming tradition, the operational aspects of CC programming have been early connected to a logical semantics based on classical logic [6], however Ruet [7] has shown that a precise logical interpretation of these languages requires linear logic (LL). The class of Linear logic Concurrent Concurrent languages (LCC) is a general extension of CC languages where the constraints are based on linear logic instead of classical logic. In addition to the classical constraint programming featured by CC, the LCC class of languages provides state changes.

The phase semantics is the natural provability semantics of linear logic. In the spirit of classical model theory, it associates formulas with values and can thus be considered the most traditional semantics of linear logic. Despite having been named "the less interesting semantics" by Girard himself [8], Fages, Ruet

and Soliman [9] have proposed an original application of the phase semantics to prove safety properties of (L)CC using the links between LL and (L)CC previously introduced by Ruet.

In this paper, we show how the phase semantics of linear logic can be applied to Constraint Handling Rules in a similar way to Fages', Ruet's and Soliman's proposal for (L)CC paradigm. In practice, to prove a safety property of a CHR program, we will exhibit a phase space and an interpretation of the program in which that particular property does hold. This result illustrates the usefulness of both the phase semantics of linear logic and linear logic semantics of CHR.

The paper is structured as follow: First we recall the basics of Constraint Handling Rules (CHR) in Sect. 2 and its LL semantics in Sect. 2.3. In Sect. 3 we present Girard's phase semantics of LL and in Sect. 4 we explain how it can be applied to prove safety properties for CHR Programs. Finally, we apply this method in one introductory and one more advanced example in Sect. 5.

## 2 Constraint Handling Rules

In CHR, we distinguish two sets of atomic constraints: The set of *built-in constraints* is handled by a predefined constraint handler. It contains at least the constraints $true$ and $false$ as well as equality $=$. The set of *user-defined constraints* is disjoint from the built-in constraints and is handled by a CHR program.

### 2.1 Syntax

**Definition 1 (CHR Program).** *A CHR program is a finite sequence of CHR rules, where a CHR rule is either:*

- *a* simplification *rule of the form:*
  $H$ `<=>` $G \mid B$
- *or a* propagation *rule of the form:*
  $H$ `==>` $G \mid B$

*where $H$ (the* head*) is a non-empty multi-set of user-defined constraints, $G$ (the* guard*) a conjunction of built-in constraints, and $B$ (the* body*) is a multiset of built-in and user-defined constraints.*

The empty guard $true$ can be omitted together with the symbol $\mid$. The notation $name$ @ $R$ gives a name to a CHR rule $R$. For the sake of simplicity, we assume without loss of generality that a variable appears at most once in the head of a rule. Furthermore, we do not allow nested disjunction in the body of a rule.

*Example 1.* The following CHR rules [1] define an ordering constraint solver. Note that equality $=$ is as usual a built-in constraint whereas we assume that $=<$ is a user-defined constraint here.

$reflexivity$ @ $X$=<$Y$ <=> $X$=$Y$ | $true$.
$antisymmetry$ @ $W$=<$X$, $Y$=<$Z$ <=> $W$=$Z$, $X$=$Y$ | $W$=$X$.
$transitivity$ @ $W$=<$X$, $Y$=<$Z$ ==> $X$=$Y$ | $W$=<$Z$.

The first rule eliminates =< constraints where both arguments are equal, the second replaces two symmetric inequality constraints by one equality constraint, and the third adds constraints in such a way as to implement transitive closure.

### 2.2 Operational Semantics

We introduce the abstract operational semantics of CHR. In this most general variant of the operational semantics, the language is inherently non-deterministic. More restricted variants of the operational semantics guarantee deterministic execution and avoidance of trivial non-determinsm but are less interesting from the theoretical point of view and with respect to concurrency.

**Definition 2 (CHR state).** *A* CHR state *is a tuple $\langle F, C\rangle$ where $F$ is a multiset of built-in and CHR constraints called* goal store *and $C$ a conjunction of built-in constraints, called* built-in store*.*

We assume without loss of generality that a *constraint theory* is a set of implications called *non-logical axioms* of the form: $\forall(C \supset D)$ where the $C$ and the $D$ are conjunction of built-in constraints.

**Definition 3 (Operational Semantics[10]).** *Given a CHR program $\mathcal{D}$ and a constraint theory $CT$, the transition relation $\rightarrow$ over states of the operational semantics, is defined inductively as the least relation satisfying the following rules:*

**Solve** $\langle\{C\} \uplus F, D\rangle \rightarrow \langle F, C \wedge D\rangle$
*if $C$ is a built-in constraint*
**Simplify** $\langle G \uplus E, D\rangle \rightarrow \langle B \uplus E, G{=}H G \wedge \wedge D\rangle$
*if ($H$ <=> $C \mid B$) is in $P$ renamed with fresh variables*
*and $CT \models D \rightarrow \exists(G{=}H \wedge C)$*
**Propagate** $\langle G \uplus E, D\rangle \rightarrow \langle B \uplus G \uplus E, G{=}H \wedge G \wedge D\rangle$
*if ($H$ ==> $C \mid B$) is in $P$ renamed with fresh variables*
*and $CT \models D \rightarrow (G{=}H \wedge C)$ then*

*Example 2.* One possible execution of the program of the previous example 1 is:

$\langle\{$Z=<X, X=<Y, Y=<Z$\}, true\rangle$
$\langle\{$X=<Z, Z=<X, X=<Y, Y=<Z$\}, true\rangle$ (**Propagate** transitivity)
$\langle\{$X=Z, X=<Y, Y=<Z$\}, true\rangle$ (**Simplify** antisymmetry)
$\langle\{$X=<Y $\wedge$ Y=<Z$\}$, X=Z$\rangle$ (**Solve**)
$\langle\{$X=Y$\}$, X=Z$\rangle$ (**Simplify** antisymmetry )
$\langle\emptyset$, X=Y $\wedge$ X=Z$\rangle$ (**Solve**)

### 2.3 Linear Logic Semantics

In this section we recall the result of [2]. The CHR rules and CHR states are translated into ILL as presented in the table 2.3.

| | |
|---|---|
| true | $true^\dagger = \mathbf{1}$ |
| non-logical axioms | $\forall(C \supset D) = !\forall(C^\dagger \multimap D^\dagger)$ |
| built-in constraints | $C^\dagger = !C$ |
| CHR constraint | $C^\dagger = C$ |
| empty multiset | $\emptyset^\dagger = \mathbf{1}$ |
| conjunction | $(C_1 \wedge \cdots \wedge C_n)^\dagger = C_1^\dagger \otimes \cdots \otimes C_n^\dagger$ |
| simplification rules | $(H \iff G \mid B)^\dagger = \forall((G^\dagger \otimes H^\dagger) \multimap \exists \bar{y}.B^\dagger)$ |
| propagation rules | $(H \implies G \mid B)^\dagger = \forall((G^\dagger \otimes H^\dagger) \multimap H^\dagger \otimes \exists \bar{y}.B^\dagger)$ |
| programs | $\{R_1, \ldots, R_n\}^\dagger = \{R_1^\dagger, \ldots, R_n^\dagger\}$ |
| with $\bar{y} = \mathrm{fv}(G, B) \setminus \mathrm{fv}(H)$ | |

**Table 1.** Translation of CHR into ILL

In the following, $CT^\dagger$ is the translation of some constraint theory $CT$ using usual Girard's translation of classical logic into linear logic.

**Theorem 1 (Soundness [2]).** *If a CHR state $T$ is derivable from another CHR state $S$ under a program $\mathcal{D}$ and a constraint theory $CT$ then the following holds:*

$$CT^\dagger, P^\dagger \vdash \forall(S^\dagger \multimap T^\dagger)$$

**Theorem 2 (Completeness [2]).** *Let $S$ and $T$ be two CHR states. If two states are such that :*

$$CT^\dagger, P^\dagger \vdash \forall(S^\dagger \multimap T^\dagger)$$

*then there exists a state $T'$ derivable from $S$ such as :*

$$CT^\dagger \vdash (S^\dagger \multimap T^\dagger)$$

## 3 Phase Semantics

Phase semantics is the natural provability semantics of linear logic [4]. It has been successfully applied by Fages et al. in order to prove safety properties of LCC programs through the logical semantics of LCC.

**Definition 4 (Phase Space).** *A phase space $\mathbf{P} = (P, \cdot, 1, \mathcal{F})$ is a commutative monoid $(P, \cdot, 1)$ together with a set $\mathcal{F}$ of subsets of $P$, whose elements are called* fact*, such that:*

– *$\mathcal{F}$ is closed under arbitrary intersection,*
– *for all $A \subset P$, for all $F \in \mathcal{F}$, $A \multimap F = \{x \in P : \forall a \in A, a \cdot x \in F\}$ is a fact.*

*A* parametrical fact *$A$ is a total function from $\mathcal{V}$ to $\mathcal{F}$ assigning to each variable $x$ a fact $A(x)$. Any fact can be seen as a constant parametrical fact, and any operation defined on fact: $(A \star B)(x) = A(x) \star B(x)$.*

*Given $A$ and $B$ two parametrical facts, we define the following facts:*

$$\begin{aligned} A\&B &= A \cap B \\ A \otimes B &= \bigcap \{F \in \mathcal{F} : A \cdot B \subset F\} \\ A \oplus B &= \bigcap \{F \in \mathcal{F} : A \cup B \subset F\} \\ \exists x.A &= \bigcap \left\{F \in \mathcal{F} : (\textstyle\bigcup_{x \in \mathcal{V}} A(x)) \subset F\right\} \\ \forall x.A &= \bigcap \left\{F \in \mathcal{F} : (\textstyle\bigcap_{x \in \mathcal{V}} A(x)) \subset F\right\} \end{aligned}$$

Here are a few notable facts: the greatest fact $\top = P$, the smallest fact $\mathbf{0}$ and $\mathbf{1} = \bigcap \{F \in \mathcal{F} : 1 \in F\}$.

**Definition 5 (Enriched Phase Space).** *An* enriched phase space *is a phase space $(P, \cdot, 1, \mathcal{F})$ together with a subset $\mathcal{O}$ of $\mathcal{F}$, whose elements are called* open facts*, such that:*

– *$\mathcal{O}$ is closed under arbitrary $\oplus$,*
– *$\mathbf{1}$ is the greatest open fact,*
– *$\mathcal{O}$ is closed under finite $\otimes$,*
– *$\otimes$ is idempotent on $\mathcal{O}$ (if $A \in \mathcal{O}$ then $A \otimes A = A$).*

*$!A$ is defined as the greatest open fact contained in $A$.*

**Definition 6 (Valuation).** *Given an enriched phase space, a* valuation *is a mapping $\eta$ from atomic ILL formulas to facts such that $\eta(\top) = \top$, $\eta(\mathbf{1}) = \mathbf{1}$ and $\eta(\mathbf{0}) = \mathbf{0}$.*

**Definition 7 (Interpretation).** *The interpretation $\eta(A)$ of a formula $A$ is defined inductively a follows:*

$$\begin{aligned} \eta(A \otimes B) &= \eta(A) \otimes \eta(B) \\ \eta(A \multimap B) &= \eta(A) \multimap \eta(B) \\ \eta(A\&B) &= \eta(A)\&\eta(B) \\ \eta(A \oplus B) &= \eta(A) \oplus \eta(B) \\ \eta(!A) &= !\eta(A) \\ \eta(\exists x.A) &= \exists x.\eta(A) \\ \eta(A) &= \eta(A) \end{aligned}$$

*We extend this interpretation to multi-sets of formulas in the obvious way by considering the coma as $\otimes$, that is to say $\eta(\emptyset) = \mathbf{1}$ and $\eta(A_1, \ldots, A_n) = \eta(A_1) \otimes \cdots \otimes \eta(A_n)$.*

This takes us to defining a notion of validity:

**Definition 8 (Validity).**

$$\mathbf{P}, \eta \models (\Gamma \vdash A) \text{ if and only if } \eta(\Gamma) \subset \eta(A)$$

$$\mathbf{P} \models (\Gamma \vdash A) \text{ if for every valuation } \eta\ \mathbf{P}, \eta \models (\Gamma \vdash A)$$

$$\models (\Gamma \vdash A) \text{ if for every phase space } \mathbf{P}\ \mathbf{P} \models (\Gamma \vdash A)$$

**Theorem 3 (Soundness [4, 11]).**

*If there is a sequent calculus proof of $\Gamma \vdash A$ then $\models (\Gamma \vdash A)$.*

**Theorem 4 (Completeness [4, 11]).**

*If $\models (\Gamma \vdash A)$ then there exists a sequent calculus proof of $\Gamma \vdash A$.*

## 4 Proving Safety Properties

As Fages et al. [9] have done for the (L)CC paradigm, we can use the phase semantics presented above to prove safety properties for CHR. Indeed, by considering CHR under the abstract operational semantics presented in Definition 3 CHR can be viewed as a subset of a Linear CC language.

Due to the soundness theorem 3 with respect to ILL, we know that:

$$\text{If } \forall CT^\dagger, \mathcal{D}^\dagger \vdash \forall(S^T \multimap T^\dagger) \text{ then } \eta(CT^\dagger, \mathcal{D}^\dagger) \subset \eta(\forall(S^T \multimap T^\dagger)).$$

By contrapositive we get that there exists a phase space $\mathbf{P}$ and a valuation $\eta$ such that

$$\text{If } \eta(CT^\dagger, \mathcal{D}^\dagger) \not\subset \eta(\forall(S^T \multimap T^\dagger)) \text{ then } \forall CT^\dagger, \mathcal{D}^\dagger \nvdash \forall(S^T \multimap T^\dagger).$$

Finally, by using the contrapositive of the soundness theorem 1 we have

$$\text{If } \forall CT^\dagger, \mathcal{D}^\dagger \nvdash \forall(S^T \multimap T^\dagger) \text{ then } S \not\rightarrow T.$$

We have hence the following proposition which allows us to reduce a problem of non-existence of a derivation between two CHR states – i.e. a *safety property* – to a problem of existence of a phase space and an interpretation of the program in which a simple inclusion is not possible. As explained in [9], only the completeness of the logical semantics is used in to prove the property. Nonetheless, the soundness theorem gives us the certitude that a such semantical proof of a true property exists.

**Proposition 1.** *Let $CT$ be a constraint theory and $\mathcal{D}$ a CHR program. To prove a safety property of the form $S \not\rightarrow T$, it is enough to prove there exists a phase space $\mathbf{P}$, a valuation $\eta$ a substitution $\sigma$ and a element $a \in \eta(S\sigma^\dagger)$ such that:*

1. *For any non-logical axiom: $\forall(C_1 \wedge \cdots \wedge C_m) \supset (D_1 \dots D_n)$ of $CT$, the inclusion $(\eta(!C_1) \otimes \cdots \otimes \eta(!C_m)) \subset (\eta(!D_1) \otimes \cdots \otimes \eta(!D_n))$ holds;*

2. *For any CHR rule* $H_1, \ldots, H_l$ <=> $G_1 \wedge \cdots \wedge G_m \mid B_1, \ldots, B_n$ *of* $\mathcal{D}$ *the inclusion* $(\eta(H_1) \otimes \cdots \otimes \eta(H_l) \otimes \eta(!G_1) \otimes \cdots \otimes \eta(!G_m)) \subset (\eta(B_1^\dagger) \otimes \cdots \otimes \eta(B_n^\dagger)))$ *holds;*
3. $a \notin \eta((T\sigma)^\dagger)$.

*Proof.* First notice that conditions *1* and *2* imply that $\mathbf{P}, \eta \models CT^\dagger, \mathcal{D}^\dagger$ and then $\mathbf{1} \in \eta(CT^\dagger, \mathcal{D}^\dagger)$. Now let us suppose that that $a \in \eta((S\sigma)^\dagger)$ and $a \notin \eta((T\sigma)^\dagger)$. Hence we infer that $\mathbf{1} \notin \eta((S\sigma)^\dagger) \multimap \eta((T\sigma)^\dagger)$. Therefore $\mathbf{1} \notin \eta(\forall((S)^\dagger) \multimap \eta((T)^\dagger))$ and then $\eta(CT^\dagger, \mathcal{D}^\dagger) \not\subset \forall\eta((S)^\dagger) \multimap \eta((T)^\dagger)$. Using the soundness theorem 3 we infer that $(CT^\dagger, \mathcal{D}^\dagger) \not\vdash \forall\eta((S)^\dagger) \multimap \eta((T)^\dagger)$. Using the soundness theorem 1, we conclude that $S \not\rightarrow T$. □

## 5 Examples

### 5.1 The Three Dining Philosophers Problem

In this example, we formulate a CHR program that implements the Dining Philosophers Problem for three philosophers. Subsequently, we use the phase semantics of linear logic to prove that from the canonic initial state, our program will never reach a state in which both philosopher #1 and philosopher #2 are eating at the same time.

**i) The Program.**

Let $\mathcal{D}$ be the following program defined under the trivial constraint theory $CT$:

$fork(1), fork(2)$ <=> $eat(1)$
$fork(2), fork(3)$ <=> $eat(2)$
$fork(3), fork(1)$ <=> $eat(3)$
$eat(1)$ <=> $fork(1), fork(2)$
$eat(2)$ <=> $fork(2), fork(3)$
$eat(3)$ <=> $fork(3), fork(4)$

**ii) Formulate the Property.**
Our goal here is to prove, using Proposition 1, the following safety property:

$$\langle true, fork(1), fork(2), fork(3)\rangle \not\rightarrow \langle C, eat(1), eat(2), H\rangle$$

where $H$ is an arbitrary multiset of constraints.

**iii) The Phase Space.**
Consider the following structure $\mathbf{P}$:

- the monoid is $\{\mathbb{N}, \cdot, 1\}$
- $\mathcal{F} = \mathcal{D}(\mathbb{N})$ (the set of parts of $\mathbb{N}$)
- $\mathcal{O} = \{\emptyset, \{1\}\}$

For such phase space, any valuation $\eta$ respects the two conditions: $\eta(\mathbf{1}) = \{1\}$ and $\eta(\top) = \mathbb{N}$.

**iv) The Valuation.**
We define $\eta$ as follows :

- $\eta(fork(1)) = \{2\}$
- $\eta(fork(2)) = \{3\}$
- $\eta(fork(3)) = \{5\}$
- $\eta(eat(1)) = \{6\}$
- $\eta(eat(2)) = \{15\}$
- $\eta(eat(3)) = \{10\}$
- $\eta(X = Y) = \begin{cases} \{1\} & \text{if } X = Y \\ \emptyset & \text{otherwise} \end{cases}$

**v) Verify the Validity of the Constraint System (condition *1*).**

We need to prove now that the constraint system is valid with respect the phase space $\mathbf{P}$ and the valuation $\eta$. In this basic example, we can suppose without loss of generality that the constraint system is the trivial one, i.e. in only contains the basic non-logical axioms for equality:

- (reflexivity) $\forall X.(true \supset X < X + 1)$
- (symmetry) $\forall XY.(X = Y \supset Y = X)$
- (transitivity) $\forall XYZ.((X = Y \wedge Y = Z) \supset X = Y)$

Firstly, note than since $\eta(X = Y)$ is an open fact, $\eta(!X = Y) = \eta(X = Y)$. For (reflexivity) note that $\eta(\mathbf{1}) = \eta(X = X) = \{1\}$, for (symmetry) note that obviously $\eta(X = Y) = \eta(Y = X)$. For (transitivity), either $X$, $Y$ and $Z$ are equal, in which case $\eta(X = Y) \otimes \eta(Y = Z) = \eta(X = Z) = \{1\}$, or at least one of them is different from the others, in which case $\eta(X = Y) \otimes \eta(Y = Z)$ equals the empty set and is therefore trivially included in $\eta(X = Z)$.

**vi) Verify the Validity of the Program (condition *2*).**

In order to prove the validity of the program we only have to notice that:

- $\eta(eat(1)) = \eta(fork(1) \otimes fork(2)) = \{6\}$
- $\eta(eat(2)) = \eta(fork(2) \otimes fork(3)) = \{15\}$
- $\eta(eat(3)) = \eta(fork(3) \otimes fork(1)) = \{10\}$

**vii) Counter-example (condition *3*).**

It can now be easily verified that:
$\eta(\langle true, fork(1), fork(2), fork(3)\rangle^\dagger) = \eta(fork(1) \otimes fork(2) \otimes fork(3)) = \{30\}$
$\eta(\langle C, eat(1), eat(2), H\rangle^\dagger) \subset \eta(eat(1) \otimes eat(2) \otimes \top) = 90 \cdot \mathbb{N}$

We deduce hence that $30 \in \eta(\langle true, fork(1), fork(2), fork(3)\rangle^\dagger)$ and $30 \notin \eta(\langle C, eat(1), eat(2), H\rangle^\dagger)$. Therefore we infer that the intended safety property $(\langle true, fork(1), fork(2), fork(3)\rangle \not\rightarrow \langle C, eat(1), eat(2), H\rangle)$ holds.

### 5.2 The $n$ Dining Philosophers Problem

In this example, we implement the Dining Philosophers Problem for an arbitrary number of philosphers and we show using the phase semantics that the program can never reach a state in which any two philosophers directly neighboring each other are eating at the same time.

**i) The Program and the Constraint Systems.**

For the sake of simplicity, we add to each CHR constraint an extra argument $N$ for the total number of philosophers. We suppose that $CT$ includes the constraint theory for natural numbers.

$eat_0$ @ $fork(M, s(M)), fork(0, s(M))$ <=> $eat(0, s(M))$.
$think_0$ @ $eat(0, s(M))$ <=> $fork(M, s(M)), fork(0, s(M))$.
$eat_{s(X)}$ @ $fork(I, N), fork(s(I), N)$ <=> $eat(s(I), N)$.
$think_{s(X)}$ @ $eat(s(I), N)$ <=> $fork(I, N), fork(s(I), N)$.
$base\_case$ @ $put\_fork(0, N)$ <=> $true$
$rec$ @ $put\_fork(s(I), N)$ <=> $fork(I, N), put\_fork(I, N)$.

**ii) Reformulate the Property.**

We want to prove that two philosophers (among $N$ philosophers) which are seated side by side cannot be eating at the same time. This can be formalized by the two following safety properties (we naturally assume there are at least two philosophers):

- case of the philosophers 0 and $N$:

$$\forall M.(\langle true, put\_fork(s(s(M)), s(s(M)))\rangle \not\rightarrow \langle C, eat(s(M), s(s(M))), eat(0, s(s(M))), H\rangle)$$

- case of the philosophers $I$ and $I + 1$:

$$\forall M.(\langle true, put\_fork(s(s(M)), s(s(M)))\rangle \not\rightarrow \langle C, eat(I, s(s(M))), eat(s(I), s(s(M))), H\rangle)$$

**iii) The Phase Space.**

We consider the same structure $\mathbf{P}$ as previously:

- the monoid is $\{\mathbb{N}, \cdot, 1\}$
- $\mathcal{F} = \mathcal{D}(\mathbb{N})$ (the set of parts of $\mathbb{N}$)
- $\mathcal{O} = \{\emptyset, \{1\}\}$

**iv) The Valuation.**

Let $\phi$ be an arbitrary bijection between natural numbers and prime numbers. Now, let us define $\eta$ as:

- $\eta(fork(I, N)) = \{\phi(I)\}$

- $\eta(eat(I,J)) = \begin{cases} \{1\} & \text{if } I = 0 \text{ and } J = 0 \\ \{\phi(M) \cdot \phi(0)\} & \text{if } I = 0 \text{ and } J = s(M) \\ \{\phi(K) \cdot \phi(s(K))\} & \text{if } I = s(K) \end{cases}$
- $\eta(put\_fork(I,N)) = \begin{cases} \left\{ \prod_{j=0}^{K} \phi(j) \right\} & \text{if } I = s(K) \\ \{1\} & \text{if } I = 0 \end{cases}$
- $\eta(I = N) = \begin{cases} \{1\} & \text{if } I = N \\ \emptyset & \text{otherwise} \end{cases}$

**v) Verify the Validity of the Constraint System (condition *1*).**

We can assume the very simple following axiomatization:

1. $\forall X(true \supset X = X)$
2. $\forall XY(X = Y \supset Y = X)$
3. $\forall XYZ(X = Y \wedge Y = Z \supset Y = X)$
4. $\forall XY.(s(X) = s(Y) \supset X = Y)$

The verification of the six first axioms is quite straightforward.

**vi) Verify the Validity of the Program (condition *2*).**

- **Validity of the rules** $eat_0$ **and** $think_0$: Notice that for any $M$ we have
$$\eta(fork(M, s(M))) \otimes \eta(fork(0, s(M))) = \{\phi(M) \cdot \phi(0)\} = \eta(eat(0, s(M))).$$
- **Validity of the rules** $eat_n$ **et** $think_n$: Notice that for any $K$ and any $N$:
$$\eta(fork(K, N)) \otimes \eta(fork(s(K), N)) = \{\phi(K) \cdot \phi(s(K))\} = \eta(eat(s(I), N)).$$
- **Validity of the rule** $base\_case$: Notice that for any for any $N$ ze have:
$$\eta(put\_fork(0, N)) = \{1\} = \eta(\mathbf{1}).$$
- **Validity of the rule** $rec$: The proof is by cases on the first argument $s(I)$ of the $put\_fork$ constraint:
  - $s(I) = s(0)$: in this case notice that for any $N$:
$$\begin{aligned} \eta(put\_fork(s(0), N)) &= \left\{ \prod_{i=0}^{0} \phi(i) \right\} = \{\phi(0)\} \otimes \{1\} \\ &= \eta(fork(0)) \otimes \eta(put\_fork(0, N)) \end{aligned}$$
  - $s(I) = s(s(K))$ for some $K$: in this case notice that for any $N$:
$$\begin{aligned} \eta(put\_fork(s(s(K)), N)) &= \left\{ \prod_{i=0}^{i=s(K)} \phi(i) \right\} = \{\phi(s(K))\} \otimes \left\{ \prod_{i=0}^{i=s(K)} \right\} \\ &= \eta(fork(s(s(K)))) \otimes \eta(put\_fork(s(K)) \end{aligned}$$

**vii) Counter-example (condition *3*).**

We now have to present two counter-examples, one for each safety property. First, we easily verify that (for any constraint multisets $C$ and $H$):

- $\eta(put\_fork(s(s(M)), s(s(M)))) = \prod_{i=0}^{i=s(M)}$
- $\eta(eat(s(M), s(s(M))), eat(0, s(s(M))), C, H) \subset \phi(0) \cdot \phi(M) \cdot \phi(s(M))^2 \cdot \mathbb{N}$
- $\eta(eat(I, s(s(M))), eat(s(I), s(s(M))), C, H) \subset \phi(I) \cdot \phi(s(I))^2 \cdot \phi(s(s(I))) \cdot \mathbb{N}$

Since $\phi(0)$, $\phi(M)$ and $\phi(s(M))$ are pairwise distinct prime numbers we have $\prod_{i=0}^{i=s(M)} \notin \phi(0) \cdot \phi(M) \cdot \phi(s(M))^2 \cdot \mathbb{N}$. Similarly since $\phi(I)$, $\phi(s(I))$ and $\phi(s(s(I)))$ are pairwise distinct prime numbers we have $\prod_{i=0}^{i=s(M)} \notin \phi(I) \cdot \phi(s(I))^2 \cdot \phi(s(s(I))) \cdot \mathbb{N}$. By Proposition 1, we prove the two safety properties.

## 6 Conclusion

Relying on the linear logic semantics of CHR, we showed that the method described by Fages, Ruet and Soliman in [9] to verify safety properties of (L)CC programs can be adapted to CHR programs as well. This adaptation is straightforward as from the point of view of its linear logic semantics, CHR can indeed be viewed as a subset of LCC. We have given a detailed explanation of our method, illustrated with two examples.

Our result provides evidence that the linear logic semantics is a useful tool for the analysis and verification of CHR programs.

While a fully automatic application of our method might not be feasible, it should be possible to significantly speed up the process with a semi-automatic system that propagates a given valuation of the facts over a program and checks whether or not this valuation proves a certain property defined by the user. This could be done with a specific finite domain solver implemented in CHR and optimized for our purpose. Such a system could spare the user the tedious and error-prone process of propagating a valuation manually.

For the future, further investigation of the apparently close relationship between CHR and (L)CC as well as the relationship between CHR and algebraic structures such as the phase semantics seems a promising approach and will hopefully produce further useful results with respect to analysis and verification of CHR programs.

**Acknowledgments**

We are grateful to Sylvain Soliman for the useful discussion about the phase semantics of linear logic.

This work was funded in part by the Madrid Regional Government under the *PROMESAS* project, the Spanish Ministry of Science under the TIN-2005-09207 *MERIT* project, and the IST program of the European Commission, under

the IST-15905 *MOBIUS*, IST-215483 *SCUBE*, and ITEA 06042 (PROFIT FIT-340005-2007-14) ESPASS projects.

Hariolf Betz is being funded by the University of Ulm under LGFG grant #0518.

## References

1. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37**(1-3) (October 1998) 95–138
2. Betz, H., Frühwirth, T.W.: A linear-logic semantics for constraint handling rules. In: Proceedings of CP 2005, 11th, Springer-Verlag (2005) 137–151
3. Betz, H.: A linear logic semantics for constraint handling rules with disjunction. In: Proceedings of the 4th Workshop on Constraint Handling Rules. (2007) 17–31
4. Girard, J.Y.: Linear logic. Theoretical Computer Science **50(1)** (1987)
5. Saraswat, V.A., Rinard, M.C.: Concurrent Constraint Programming. In: POPL'90: Proceedings of the 17th ACM Symposium on Principles of Programming Languages. (1990)
6. Saraswat, V.A., Rinard, M.C., Panangaden, P.: Semantic foundations of concurrent constraint programming. In: POPL'91: Proceedings of the 18th ACM Symposium on Principles of Programming Languages. (1991)
7. Ruet, P.: Logical semantics of concurrent constraint programming. In: Proceedings of CP'96, 2nd International Conference on Principles and Practice of Constraint Programming, Springer-Verlag (1996)
8. Girard, J.Y.: Linear logic: its syntax and semantics. In: Proceedings of the workshop on Advances in linear logic, Cambridge University Press (1995) 1–42
9. Fages, F., Ruet, P., Soliman, S.: Linear concurrent constraint programming: operational and phase semantics. Information and Computation **165**(1) (February 2001) 14–41
10. Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Springer-Verlag (February 2003)
11. Okada, M.: Girard's phase semantics and a higher-order cut-elimination proof. Technical report, Institut de Mathématiques de Luminy (1994)

# A Tale of Histories

Peter Van Weert*

Department of Computer Science, K.U.Leuven, Belgium
Peter.VanWeert@cs.kuleuven.be

**Abstract.** Constraint Handling Rules (CHR) is an elegant, high-level programming language based on multi-headed, forward chaining rules. A distinguishing feature of CHR are propagation rules. To avoid trivial non-termination, CHR implementations ensure a CHR rule is applied at most once with the same combination of constraints by maintaining a so-called propagation history. The performance impact of this history is often significant. We introduce two optimizations to reduce or even eliminate this overhead, and evaluate their implementation in two state-of-the-art CHR systems.

## 1 Introduction

Constraint Handling Rules (CHR) [1, 2] is a high-level committed-choice CLP language, based on multi-headed, guarded multiset rewrite rules. Originally designed for the declarative specification of constraint solvers, it is increasingly used for general purposes, in a wide range of applications. Efficient implementations exist for several host languages, including Prolog [3, 4], Haskell, and Java [5].

An important, distinguishing feature of CHR are *propagation rules*. Unlike traditional rewrite rules, propagation rules do not remove the constraints matched by their head. They only add extra, implied constraints. Logically, a propagation rule corresponds to an implication.

The formal study of properties such as confluence and termination, led to the extension of CHR's operational semantics with a *token store* [1]. The token store contains a token for every constraint combination that may match a propagation rule. Each time a propagation rule is applied, the corresponding token is removed. Trivial non-termination is thus avoided by applying a propagation rule at most once with the same combination of constraints.

Practical implementations of CHR use the dual notion of a token store, called a *propagation history*; *history* for short [6, 3–5]. A history contains a tuple for each constraint combination that already fired a rule. A rule is only applied with some constraint combination, if the history does not contain the corresponding tuple. This is also reflected in more recent CHR operational semantics [7].

The implementation and optimization of propagation histories never received much attention [6, 4]. Our results show however that the propagation history can have a significant impact on both space and time performance. This paper

* Research Assistant of the Research Foundation – Flanders (FWO-Vlaanderen).

constitutes a first attempt to resolve this apparent discrepancy. We introduce two novel optimization techniques that either reduce or eliminate the overhead associated with propagation history maintenance

**Contributions and Overview**

- In Section 3 we explore the design space for the implementation of propagation histories. We show why implementing a history efficiently is challenging, and review some approaches taken by existing CHR systems. We then introduce an optimization for two-headed propagation rules.
- Section 4 introduces an innovative optimization that eliminates the need for maintaining a propagation history for all *non-reactive* CHR rules. This important class of CHR rules covers the majority of rules found in general-purpose CHR programs. We prove that the optimization is correct with respect to CHR's refined operational semantics [7].
- We implemented these optimizations in two state of the art CHR implementations, K.U.Leuven CHR [4, 8] for SWI-Prolog, and K.U.Leuven JCHR for Java [5]. Section 5 reports on the significant performance gains.

## 2 Preliminaries

To make this paper relatively self-contained, this section briefly reviews CHR's basic syntax and operational semantics. Gentler introductions are found for instance in [6, 1, 4].

### 2.1 CHR Syntax

CHR is embedded in a host language $\mathcal{H}$. A *constraint type* $c/n$ is denoted by a functor/arity pair; a *constraint* $c(x_1, \ldots, x_n)$ is an atom constructed from these predicate symbols, and a list of arguments $x_i$, instances of data types offered by $\mathcal{H}$. Two classes of constraints exist: *built-in constraints*, solved by an underlying constraint solver of the host $\mathcal{H}$, and *CHR constraints*, handled by a CHR program. Many CHR systems support type and mode declarations for the arguments of CHR constraints. A CHR program $\mathcal{P}$, also called a *CHR handler*, is a sequence of CHR rules. The generic syntactic form of a CHR rule is:

$$\rho \ @ \ H_k \setminus H_r \Leftrightarrow G \mid B$$

The rule's name $\rho$ uniquely identifies a rule. The *head* consists of two conjunctions of CHR constraints, $H_k$ and $H_r$. Their conjuncts are called *occurrences* (*kept* and *removed occurrences* resp.). If $H_k$ is empty, the rule is a *simplification rule*. If $H_r$ is empty, the rule is a *propagation rule* and the symbol '$\Rightarrow$' is used instead of '$\Leftrightarrow$'. If both are non-empty, the rule is a *simpagation* rule. Either $H_k$ or $H_r$ has to be non-empty. The *guard* $G$ is a conjunction of built-in constraints. If '$G \mid$' is omitted, it is considered to be '`true` $\mid$'. The rule's *body* $B$, finally, is a conjunction of CHR and built-in constraints.

```
reflexivity  @ leq(X, X) ⇔ true.
idempotence  @ leq(X, Y) \ leq(X, Y) ⇔ true.
antisymmetry @ leq(X, Y), leq(Y, X) ⇔ X = Y.
transitivity @ leq(X, Y), leq(Y, Z) ⇒ leq(X, Z).
```

**Fig. 1.** LEQ, a CHR program for the less-than-or-equal constraint.

*Example 1.* Fig. 1 shows a classic example CHR program, LEQ. It defines one CHR constraint, a less-than-or-equal constraint, using four CHR rules. All three kinds of rules are present. The constraint arguments are logical variables. The handler uses one built-in constraint, namely equality. If the *antisymmetry* rule is applied, its body adds a new built-in constraint to the built-in equality solver provided by the host environment. The body of the *transitivity* propagation rule adds a new CHR constraint, which is handled by the CHR program itself.

*Head Normal Form* In the Head Normal Form of a CHR program $\mathcal{P}$, denoted HNF($\mathcal{P}$), a variable occurs at most once in rule heads. For instance, in HNF(LEQ), the normalized form of the *transitivity* rule from Fig. 1 is:

```
transitivity @ leq(X, Y), leq(Y1, Z) ⇒ Y = Y1 | leq(X, Z).
```

### 2.2 The Refined Operational Semantics

The behavior of most current CHR implementations is formally captured by the refined operational semantics [7], commonly denoted as $\omega_r$. The $\omega_r$ semantics is formulated as a state transition system, in which *transition rules* define the relation between subsequent *execution states*. The version presented here follows [6, 4], and is a slight modification from the original specification [7].

*Notation* Sets, multisets and sequences (ordered multisets) are defined as usual. We use $S[i]$ to denote the $i$'th element of a sequence $S$, $+\!\!+$ for sequence *concatenation*, and $[e|S]$ to denote $[e]+\!\!+S$. The *disjoint union* of sets is defined as follows: $\forall X, Y, Z : X = Y \sqcup Z \Leftrightarrow X = Y \cup Z \wedge Y \cap Z = \emptyset$. For a logical expression $X$, $vars(X)$ denotes the set of unquantified variables, and $\pi_V(X) \Leftrightarrow \exists v_1, \ldots, v_n : X$ with $\{v_1, \ldots, v_n\} = vars(X) \setminus V$. The meaning of built-in constraints is assumed determined by $\mathcal{D}_\mathcal{H}$, a consistent (first order logic) built-in constraint theory.

*Execution States* An execution state of $\omega_r$ is a tuple $\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$. The *execution stack* $\mathbb{A}$ is a sequence, used to treat constraints as procedure calls. Its function is explained in more detail below. The CHR constraint *store* $\mathbb{S}$ is a set of *identified* CHR constraints. An *identified* CHR constraint $c\#i$ is a CHR constraint $c$ associated with a unique *constraint identifier* $i$. The two connated mapping functions, $chr(c\#i) = c$ and $id(c\#i) = i$, are extended to sequences and sets in the obvious manner. The constraint identifiers are used to distinguish otherwise identical constraints ($chr(\mathbb{S})$ is a *multiset* of constraints). The counter $n$ represents the next free CHR constraint identifier. The *built-in constraint store* $\mathbb{B}$ is

**1. Solve** $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle S +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ if $b$ is a built-in constraint and $S \subseteq \mathbb{S}$ such that $\forall c \in S : vars(c) \not\subset fixed(\mathbb{B})$ and $\forall H \subseteq \mathbb{S} : (\exists K, R : H = K +\!\!+ R \wedge \exists \rho \in \mathcal{P} : \neg appl(\rho, K, R, \mathbb{B}) \wedge appl(\rho, K, R, b \wedge \mathbb{B})) \rightarrow (S \cap H \neq \emptyset)$.

---

**2. Activate** $\langle [c|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle [c\#n : 1|\mathbb{A}], \{c\#n\} \sqcup \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_{n+1}$ if $c$ is a CHR constraint (which has not yet been active or stored in $\mathbb{S}$).

---

**3. Reactivate** $\langle [c\#i|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle [c\#i : 1|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ if $c$ is a CHR constraint (re-added to $\mathbb{A}$ by a **Solve** transition but not yet active).

---

**4. Simplify** $\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle B +\!\!+ \mathbb{A}, K \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T}'\rangle_n$ with $\mathbb{S} = \{c\#i\} \sqcup K \sqcup R_1 \sqcup R_2 \sqcup S$, if the $j$-th occurrence of $c$ in $\mathcal{P}$ occurs in rule $\rho$, and $\theta$ is a matching substitution such that $appl(\rho, K, R_1 +\!\!+ [c\#i] +\!\!+ R_2, \theta, \mathbb{B}) = B$.
Let $t = (\rho, id(K +\!\!+ R_1) +\!\!+ [i] +\!\!+ id(R_2))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

---

**5. Propagate** $\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle B +\!\!+ [c\#i : j|\mathbb{A}], \mathbb{S} \setminus R, \theta \wedge \mathbb{B}, \mathbb{T}'\rangle_n$ with $\mathbb{S} = \{c\#i\} \sqcup K_1 \sqcup K_2 \sqcup R \sqcup S$, if the $j$-th occurrence of $c$ in $\mathcal{P}$ occurs in rule $\rho$, and $\theta$ is a matching substitution such that $appl(\rho, K_1 +\!\!+ [c\#i] +\!\!+ K_2, R, \theta, \mathbb{B}) = B$.
Let $t = (\rho, id(K_1) +\!\!+ [i] +\!\!+ id(K_2 +\!\!+ R))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

---

**6. Drop** $\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ if $c$ has no $j$-th occurrence in $\mathcal{P}$.

---

**7. Default** $\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle [c\#i : j+1|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ if the current state cannot fire any other transition.

**Fig. 2.** The transition rules of the refined operational semantics $\omega_r$.

an abstract logical conjunction of built-in constraints, modeling all constraints passed to the underlying solvers. The *propagation history* $\mathbb{T}$, finally, is a set of tuples, each recording a sequence of constraint identifiers of CHR constraints that fired a rule, and the unique name of that rule.

Given an initial query $Q$, a sequence (conjunction) of built-in and host language constraints, an *initial execution state* is of the form $\langle Q, \emptyset, \texttt{true}, \emptyset\rangle_1$.

*Transition Rules* The transition rules of $\omega_r$ are listed in Fig. 2. The top-most element of $\mathbb{A}$ is called the *active constraint*. Each newly added CHR constraint initiates a search for partner constraints that match the heads of the rules in an **Activate** transition. A built-in constraint is passed to the underlying constraint solver in a **Solve** transition. If the newly added built-in constraint may affect the outcome of guards, similar searches for applicable rules are initiated for the affected CHR constraints. Constraints whose variables are all fixed are never reactivated; formally:

**Definition 1.** *A variable $v$ is* fixed *by a conjunction of constraints $B$, denoted $v \in fixed(B)$, if and only if $\mathcal{D}_{\mathcal{H}} \models \pi_{\{v\}}(B) \wedge \pi_{\{\theta(v)\}}(B) \rightarrow v = \theta(v)$ for arbitrary substitution $\theta$.*

The order in which occurrences are traversed is fixed by $\omega_r$: an active constraint tries its occurrences in a CHR program in a top-down, right-to-left order. To realize this order in $\omega_r$, identified constraints on the execution stack are *occurrenced* (in **Activate** and **Reactivate** transitions). An *occurrenced* identified

CHR constraint $c\#i:j$ indicates that only matches with the $j$'th occurrence of $c$'s constraint type are considered when the constraint is active.

Each active constraint traverses its different occurrences by a sequence of **Default** transitions, followed by a **Drop** transition. During this traversal all applicable rules are fired (i.e. **Propagate** and **Simplify** transitions). The applicability of a CHR rule is defined as follows:

**Definition 2.** *Given a conjunction of built-in constraints $\mathbb{B}$, a rule $\rho$ is* applicable *with sequences of identified CHR constraints $K$ and $R$, denoted $appl(\rho, K, R, \mathbb{B})$, if and only if a matching substitution $\theta$ exists for which $appl(\rho, K, R, \theta, \mathbb{B})$ is defined. The latter partial function is defined as $appl(\rho, K, R, \theta, \mathbb{B}) = B$ if and only if $K \cap R = \emptyset$ and, renamed apart, $\rho$ is of the form ($H_k$ or $H_r$ may be empty):*

$$\rho \ @ \ H_k \setminus H_r \Leftrightarrow G \mid B$$

*such that $chr(K) = \theta(H_k)$, $chr(R) = \theta(H_r)$ and $\mathcal{D}_\mathcal{H} \models \mathbb{B} \rightarrow \pi_{vars(\mathbb{B})}(\theta \wedge G)$.*

As with a procedure, when a rule fires, other constraints (its body) are executed, and execution does not return to the original active constraint *until* after these calls have finished. By putting the body on the activation stack, the different conjuncts of the body are solved (for built-in constraints) or activated (for CHR constraints) in a left-to-right order. This approach corresponds closely to that of the stack-based programming languages to which CHR is compiled.

*Derivations* Execution proceeds by exhaustively applying transitions. Formally, a derivation $D$ is a sequence of states, with $D[1]$ a valid initial execution state for some query $Q$, and $D[i] \rightarrowtail_\mathcal{P} D[i+1]$ for all subsequent states $D[i]$ and $D[i+1]$. We also say these transitions $D[i] \rightarrowtail_\mathcal{P} D[i+1]$ are transitions of $D$. The common notational abbreviation $\sigma_1 \rightarrowtail^\star_\mathcal{P} \sigma_n$ denotes a *finite* derivation $[\sigma_1, \ldots, \sigma_n]$.

## 3 Propagation History Implementation

As stated also in [6, Section 4.3.4], a propagation history is very easy to implement naively, but quite challenging to implement efficiently. Obviously, tuples have to be stored in some efficient data structure, e.g. a balanced tree or a hash table. Naively implemented, tuples are only added to the propagation history, but never removed. Note that this is also the case in the $\omega_r$ formalism (cf. Section 2.2). This potentially leads to unbounded memory use.

The main challenge is thus to avoid this memory problem, with minimal overhead. All tuples referring to removed constraints are redundant. Formally, for a state $\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$, these are all tuples not in $live(\mathbb{T}, \mathbb{S}) = \{(\rho, I) \in \mathbb{T} \mid I \subseteq id(\mathbb{S})\}$. Practice shows that eagerly removing redundant tuples after each constraint removal is not feasible due to time or space overheads. CHR implementations therefore commonly use ad-hoc garbage collection techniques, which may result in excessive memory use, but perform adequately in practice.

One technique is to lazily remove redundant tuples during history checks (see [6]). A second technique is denoted *distributed propagation history* maintenance

(see [4]). With this technique, no global propagation history is maintained. Instead, the runtime representation of each individual CHR constraint contains (a subset of) the history tuples they occur in. When a constraint is removed, the corresponding part of the propagation history is thus removed as well. Both techniques could easily be combined.

We refer to [6, 4] for some more details on the implementation of propagation histories in current CHR systems. Many design choices, however, are not fully covered by these theses:

- Is one global history maintained, or one history per rule?
- Is the distributed history information stored in all constraints of the matching combination, or only in one of the partners? In the latter case, is the active constraint used, or the constraint matching some fixed occurrence?
- In which cases are more eager garbage collection techniques feasible?
- How to exploit functional dependencies?

In the following subsection we introduce an improved technique to maintain the propagation history of two-headed propagation rules.

### 3.1 Two-headed Propagation Rules

For two-headed propagation rules, a distributed propagation history can be implemented more efficiently. Assume that, if there are multiple propagation rules, a separate history is maintained per rule, as is the case e.g. in the K.U.Leuven JCHR system [5]. By default, history tuples for a two-headed rule contain two constraint identifiers. It is however more efficient to simply store, in each constraint, the identifiers of all partner constraints it fired with whilst active. This avoids the creation of tuples, and allows for more efficient hash tables. We refer to Section 5 for empirical results.

Care must be taken when both heads are occurrences of the same constraint type, as for instance in the *transitivity* rule of Example 1. One possibility is to maintain a separate history per occurrence. Another trick is to use negated constraint identifiers if the the active constraint matches one of the occurrences.

With a similar reasoning, a reduction of the tuple size for all propagation rules is possible. Experiments only showed negligible performance gains though.

## 4 Non-reactive CHR Rules

In this section we consider *non-reactive CHR rules*, i.e. rules that are never matched by a reactivated CHR constraint. We will show that for this important class of rules no propagation history has to be maintained.

*Example 2.* Consider the following common CHR pattern to compute the sum of the arguments of `elem/1` constraints:

```
sum, elem(X) ⇒ sum(X).
sum ⇔ true.
sum(X), sum(Y) ⇔ sum(X+Y).
```

If the type or mode declarations of the `elem/1` constraint specify that its argument is always fixed, say a (ground) integer value, then `elem/1` constraints are never reactivated under $\omega_r$. As the `sum/0` constraint is clearly also never reactivated, the first rule is thus never matched by a reactivated CHR constraint.

Formally, non-reactive CHR constraints and rules are defined as follows:

**Definition 3.** *A CHR constraint type $c/n$ is* non-reactive *in a program $\mathcal{P}$ under a refined operational semantics $\omega_r^\star$ if and only if for all $\omega_r^\star$ derivations $D$ with that program, for all **Solve** transitions in $D$ of the form*

$$\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle S +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$$

*the set of reconsidered constraints $S$ does not contain constraints of type $c/n$. A CHR rule $\rho \in \mathcal{P}$ is* non-reactive *if and only if all constraint types that occur in its head are non-reactive in $\mathcal{P}$.*

Under the $\omega_r$ semantics as defined in Section 2.2, only *fixed*, or *ground*, CHR constraint types are non-reactive. Formally, a CHR constraint type $c/n$ is fixed iff for all CHR constraints $c'$ of type $c/n$, $vars(c') \subseteq fixed(\emptyset)$ (see Definition 1). A CHR compiler derives which constraints are fixed from their mode declarations, or using static groundness analysis [9]. Both constraints in Example 2, for instance, are fixed.

A substantially larger class of CHR programs, however, can be made non-reactive by a slight modification of the refined operational semantics.

*Example 3.* Suppose the type or mode information implies the first argument of `fib/2` constraints is always fixed. The second argument on the other hand can be a free (logical) variable:

```
fib(N,M1) \ fib(N,M2) ⇔ M1 = M2.
fib(N,M) ⇒ N ≤ 1 | M = 1.
fib(N,M) ⇒ N > 1 | fib(N-1,M1), fib(N-2,M2), M = M1 + M2.
```

For this handler, a `fib/2` constraint does not have to be reactivated when a built-in constraint is added. Indeed: because there are no guards on this argument, no additional rules become applicable by constraining it further.

Using constraints unbound, unguarded arguments to retrieve computation results is very common in CHR. These constraints should not be reactivated. Unfortunately, this is insufficiently specified in the standard $\omega_r$ semantics. We therefore propose a semantical refinement, based on the concept of *anti-monotonicity* [10]. Anti-monotonicity generalizes both fixed and unguarded constraint arguments:

**Definition 4.** *A conjunction of built-in constraints $B$ is* anti-monotone *in a set of variables $V$ if and only if:*

$$\forall B_1, B_2 : (\pi_{vars(B)\setminus V}(B_1 \wedge B_2)) \Leftrightarrow (\pi_{vars(B)\setminus V}(B_1)) \\ \Rightarrow (\mathcal{D}_{\mathcal{H}} \not\models B_1 \rightarrow B) \Rightarrow (\mathcal{D}_{\mathcal{H}} \not\models B_1 \wedge B_2 \rightarrow B)$$

**Definition 5.** *A CHR program $\mathcal{P}$ is* anti-monotone *in the i'th argument of a CHR constraint type $c/n$, if and only if for every occurrence $c(x_1, \ldots, x_i, \ldots, x_n)$ in HNF($\mathcal{P}$), the guard of the corresponding rule is anti-monotone in $\{x_i\}$.*

Based on these definitions, the *anti-monotony-based delay avoidance* optimization reduces the amount of needlessly reactivated constraints [10]. Concretely, let $delay\_vars_{\mathcal{P}}(c)$ denote the set of variables that occur in the arguments of an (identified) CHR constraint $c$ in which $\mathcal{P}$ is not anti-monotone, then the **Solve** transition of $\omega_r$ (cf. Fig. 2) can be replaced with:

> **1. Solve'** $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle S +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ if $b$ is a built-in constraint and $S \subseteq \mathbb{S}$ such that $\forall c \in S : \mathbf{delay\_vars}_{\mathcal{P}}(c) \not\subset fixed(\mathbb{B})$ and $\forall H \subseteq \mathbb{S} : (\exists K, R : H = K +\!\!+ R \wedge \exists \rho \in \mathcal{P} : \neg appl(\rho, K, R, \mathbb{B}) \wedge appl(\rho, K, R, b \wedge \mathbb{B})) \rightarrow (S \cap H \neq \emptyset)$.

The resulting semantics, denoted $\omega_r'$, is an instance of $\omega_r$[1].

Clearly, the following properties hold for any CHR program $\mathcal{P}$:

- If the CHR constraint type $c/n$ is fixed, i.e. if $c/n$ is non-reactive in $\mathcal{P}$ under $\omega_r$, then $\mathcal{P}$ is anti-monotone in all $n$ arguments of $c/n$.
- If $\mathcal{P}$ is anti-monotone in all $n$ arguments of $c/n$, then that CHR constraint type is non-reactive in $\mathcal{P}$ under $\omega_r'$.

We now show how the maintenance of a propagation history for non-reactive CHR rules can be avoided. The central observation is that when a non-reactive rule is fired, the active constraint is more recent than its partner constraints:

**Lemma 1.** *Let $\mathcal{P}$ be an arbitrary CHR program, with $\rho \in \mathcal{P}$ a non-reactive rule, and $D$ an arbitrary derivation with this program. Then for each* **Simplify** *or* **Propagate** *transition in $D$ of the form*

$$\langle [c\#i : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle \mathbb{A}', \mathbb{S}', \mathbb{B}', \mathbb{T} \sqcup \{(\rho, I_1 +\!\!+ [i] +\!\!+ I_2)\} \rangle_n \qquad (1)$$

*the following holds: $\forall i' \in I_1 \cup I_2 : i' < i$.*

*Proof.* Assume $i' = \max(I_1 \sqcup I_2)$ with $i' \geq i$. By Definition 2 of rule applicability, $i' \neq i$, and $\exists c'\#i' \in \mathbb{S}$. This $c'\#i'$ partner constraint must have been stored in an **Activate** transition. Since $i' = \max(I_1 \sqcup \{i\} \sqcup I_2)$, in $D$, this transition came *after* the **Activate** transitions of all other partners, including $c\#i$. In other words, all constraints in the matching combination of transition (1) were stored prior to the activation of $c'\#i'$. Also, in (1), $c\#i$ is back on top of the activation stack. Because $c$ is non-reactive, and thus never put back on top by a **Reactivate** transition, the later activated $c'\#i'$ must have been removed from the stack in a **Drop** transition. This implies that all applicable rules matching $c'$ must have fired. As all required constraints were stored (cf. supra), this includes the application of $\rho$ in (1). By contradiction, our assumption is false, and $i' < i$. □

[1] The **Solve'** transition presented here differs from the one proposed in [10]. As shown in Appendix A, the latter version is not entirely correct. The appendix further provides a correctness proof for our version, and shows that it is stronger than that of [10].

Let $\omega_r''$ denote the semantics obtained by replacing the phrase

> Let $t = (\rho, id(H_1) +\!\!+ [i] +\!\!+ id(H_2))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

in the **Simplify** and **Propagate** transitions of $\omega_r'$ with

> If $\rho$ is non-reactive, then $\forall i' \in id(H_1 \cup H_2) : i' < i$ and $\mathbb{T}' = \mathbb{T}$. Otherwise, let $t = (\rho, id(H_1) +\!\!+ [i] +\!\!+ id(H_2))$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

To avoid trivial non-termination where the same combination of constraints fires a propagation rule infinitely many times, we also assume the following property to hold for $\omega_r''$:

**Definition 6 (Duplicate-free Propagation).** *For all derivations $D$ of a CHR program $\mathcal{P}$ where the $j$'th occurrence of $c$ is kept, if the following holds:*

- $\sigma_1 \rightarrowtail_{\mathcal{P}} \sigma_2 \rightarrowtail_{\mathcal{P}}^{\star} \sigma_1' \rightarrowtail_{\mathcal{P}} \sigma_2'$ *is part of* $D$
- $\sigma_1 = \langle [c\#i : j | \mathbb{A}], \mathbb{S}, \ldots \rangle_{\_}$ *and* $\sigma_1' = \langle [c\#i : j | \mathbb{A}], \mathbb{S}', \ldots \rangle_{\_}$
- $\sigma_1 \rightarrowtail_{\mathcal{P}} \sigma_2$ *is a* ***Propagate*** *transition applied with constraints* $H \subseteq \mathbb{S}$
- $\sigma_1' \rightarrowtail_{\mathcal{P}} \sigma_2'$ *is a* ***Propagate*** *transition applied with constraints* $H' \subseteq \mathbb{S}'$
- *between* $\sigma_2$ *and* $\sigma_1'$ *no* ***Default*** *transition occurs of the form*

$$\sigma_2 \rightarrowtail_{\mathcal{P}}^{\star} \langle [c\#i : j | \mathbb{A}], \ldots \rangle_{\_} \rightarrowtail_{\mathcal{P}} \langle [c\#i : j+1 | \mathbb{A}], \ldots \rangle_{\_} \rightarrowtail_{\mathcal{P}}^{\star} \sigma_1'$$

*then* $H \neq H'$.

This property, in combination with Lemma 1, allows us to show that $\omega_r'$ and $\omega_r''$ are equivalent:

**Theorem 1.** *Define the mapping function $\alpha$ as follows:*

$$\alpha(\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n) = \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \{(\rho, I) \in \mathbb{T} \mid \rho \text{ is a reactive CHR rule}\} \rangle_n$$

*If $D$ is an $\omega_r'$ derivation, then $\alpha(D)$ is an $\omega_r''$ derivation. Conversely, if $D$ is an $\omega_r''$ derivation, then there exists an $\omega_r'$ derivation $D'$ such that $\alpha(D) = D'$.*

*Proof.* If $D$ is an $\omega_r'$ derivation, then $\alpha(D)$ is an $\omega_r''$ derivation by Lemma 1.

For the reverse direction, let $D$ be an $\omega_r''$ derivation, and $D'$ the derivation obtained from $D$ by adding the necessary tuples to the propagation history. That is, for each **Propagate** or **Simplify** transition in $D$ of the form

$$\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle B +\!\!+ \mathbb{A}, \mathbb{S}', \mathbb{B}, \mathbb{T} \rangle_n$$

the corresponding transition in $D'$ becomes of the form

$$\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle B +\!\!+ \mathbb{A}, \mathbb{S}', \mathbb{B}, \mathbb{T} \cup \{(\rho, I)\} \rangle_n \tag{2}$$

We treat the history $\mathbb{T}$ to be a multiset here, because otherwise possible duplicates would disappear unnoticed. All **Propagate** and **Simplify** transitions in $D'$ now have form (2). It suffices to show that for all these transitions $(\rho, I) \notin \mathbb{T}$.

First, we show that Lemma 1 still holds for the derivation $D$. That is, for all transitions of $D$ of form (2), if the active constraint matched the $k$'th occurrence in $\rho$'s head, then $I[k] = max(I)$. By definition of $\omega_r''$, this is true for the tuples

```
procedure up_to(U)#ID : 2
  foreach fib(N,M2)#ID2 in ...
    foreach fib(N-1,M1)#ID1 in ...
      if N < U
        if ID < ID1 and ID < ID2
          ⋱
```

(a) Unoptimized

```
procedure up_to(U)#ID : 2
  foreach fib(N,M2)#ID2 in ...
    if ID < ID2 and N < U
      foreach fib(N-1,M1)#ID1 in ...
        if ID < ID1
          ⋱
```

(b) After *Loop Invariant Code Motion*

**Fig. 3.** Pseudocode for the second occurrence of the `up_to/1` constraint of Example 4.

that were not added to the history in the original derivation $D$. For those added in both $D$ and $D'$, this also holds by definition of $\omega_r''$ and Lemma 1.

Suppose, for some transition of form (2), that $(\rho, I) \in \mathbb{T}$, and that the active constraint matched the $k$'th occurrence of $\rho$. Then $I[k] = max(I)$. Moreover, when the $(\rho, I)$ tuple was first added to the history, by uniqueness of constraint identifiers, the active constraint was the same constraint as active in the considered constraint. As propagation is duplicate-free in $D$, and the active constraint is non-reactive, this is not possible. □

This theorem shows the correctness of replacing the propagation history of non-reactive CHR rules with more efficient constraint identifier comparisons. The next subsection shows how this optimization can be implemented in typical $\omega_r$-based CHR implementations.

### 4.1 Implementation and Further Optimizations

The standard CHR compilation scheme (see e.g. [6, 3, 4]) generates, for each occurrence, a nested iteration that looks for matching partner constraints for the active constraint. If the active constraint is not removed, all partner constraint iterators are suspended until the body is fully executed. Afterwards, the nested iteration is simply resumed to find more matching combinations.

*Example 4.* The following handler, called FIBBO, performs a bottom-up computation of all Fibonacci numbers up to a given number (all arguments are fixed):

```
up_to(U) ⇒ fib(0,1), fib(1,1).
up_to(U), fib(N-1,M1), fib(N,M2) ⇒ N < U | fib(N+1,M1+M2).
```

If an `up_to(U)` constraint is told, the first rule propagates two `fib/2` constraints. After this, the second rule propagates all required `fib/2` constraints, each time with a `fib/2` constraint as the active constraint. When, finally, the `up_to(U)` constraint reaches its second occurrence, some mechanism is required to prevent the second rule to propagate everything all over again.

A propagation history would require $\mathcal{O}(U)$ space. Because all constraints are non-reactive, however, no propagation history is maintained. Instead, constraint identifiers are simply compared. Fig. 3(a) shows the generated code for the second occurrence of the `up_to/1` constraint.

| | SWI | | JCHR | |
|---|---|---|---|---|
| | tree | 2-hash | hash | 2-hash |
| EQ(35) | 3,465 | N/A[2] | 47 | 37 (79%) |
| LEQ(70) | 3,806 | 2,866 (75%) | 85 | 65 (76%) |

**Table 1.** Benchmark results for the EQ and LEQ benchmarks.

| | SWI | | JCHR | | |
|---|---|---|---|---|---|
| | tree | non-react | hash | non-react | non-react+ |
| WFS(200) | 2,489 | 2,143 (86%) | 71 | 67 (94%) | 67 (94%) |
| FIBBO(1000) | 15,929 | 4,454 (28%) | 70 | 67 (95%) | 21 (30%) |
| FIBBO(2000) | 61,290 | 17,704 (29%) | 206 | 275 (133%) | 90 (44%) |
| FIBBO(3000) | *timeout* | *timeout* | 542 | 464 (85%) | 153 (28%) |

**Table 2.** Benchmark results for the WFS and FIBBO benchmarks.

If none of the iterators return candidate partner constraints more than once, propagation is guaranteed to be duplicate-free (see Definition 6). Most iterators used by CHR implementations obey this property. If not, a temporary history can be maintained whilst the active constraint is considering an occurrence.

*Loop-invariant Code Motion* Lemma 1 not only applies to propagation rules, but also to simplification and simpagation rules. Whilst maintaining a history for non-propagation rules is pointless, comparing partner constraint identifiers is not. As shown in Fig. 3(b), the standard *Loop-invariant Code Motion* optimization can be extended to include not only guards (e.g. `N < U`), but also identifier comparisons. For multi-headed CHR rules — including simplification and simpagation rules — this may considerably prune the search space of candidate partner constraints. Moreover, if an iterator returns constraints in ascending order of identifiers, the corresponding (nested) iteration can be stopped early.

## 5 Evaluation

We implemented the optimizations presented in this paper in the K.U.Leuven CHR system [4, 8] for SWI-Prolog, and in the K.U.Leuven JCHR system [5] for Java. The benchmark results are given in Tables 1 and 2. For each system, the first column gives the reference timings: for SWI this is a tree-based propagation history, for JCHR a hash-based history. Both systems use distributed history maintenance (see Section 3). The *2-hash* and *non-react* columns give timings using the optimization for two-headed propagation rules given in Section 3, and the optimization for non-reactive CHR rules of Section 4 respectively. For the *non-react+* measurements the non-reactiveness optimization was combined with loop invariant code motion (currently only implemented in JCHR).

[2] In the current SWI implementation, the history of a two-headed propagation rule is only optimized if there are no other propagation rules in the program. In JCHR, this is not relevant, as JCHR maintains a separate history per rule.

For both optimizations significant performance gains are measured. Note that the improved timings in Table 1 for the SWI-Prolog system may be due to moving from a tree-based history to a hash-based one. For JCHR, however, this is definitely not the case, showing the relevance of the improved data structure. Table 2 contains one surprising timing for the FIBBO($N$) benchmark in JCHR: even though identifier comparisons are cheaper than checking a propagation histories, for $N = 2000$, the performance nevertheless worsened. Detailed profiling showed that this is due to unpredictable behavior of the JVM's garbage collector.

For non-reactive rules, space complexity is furthermore optimal: propagation histories no longer consume space at all. The complexity for the history of the FIBBO handler, for instance, is improved from linear to constant (see Example 4).

## 6 Related Work, Conclusions and Future Work

**Related Work** The propagation history contributes to significant performance issues when implementing CHR in a tabling environment [11]. Based on a similar approach explored in [11], an alternative CHR semantics is proposed in [12]. Being set-based, this semantics addresses the trivial non-termination problem without the use of a propagation history. It would be interesting to see whether these results can be transferred to CHR without abandoning its common multiset semantics (see also Future Work).

In [6], a simple analysis is presented to eliminate the propagation histories for certain fixed CHR constraints. Advanced CHR systems such as [8, 5] implement more powerful versions of this analysis, extended towards non-reactive CHR constraints, or made more accurate by abstract interpretation [9]. Our results in Section 4, however, considerably reduce the benefits of these complex analyses, as comparing constraint identifiers is much cheaper than maintaining a history.

**Conclusions** We showed that maintaining a propagation history comes at a considerable runtime cost, both in time and in space. We introduced two optimizations to reduce or eliminate this overhead. We showed that for two-headed propagation rules more efficient data structures can be used. This is interesting, as rules with more than two heads are relatively rare. We then argued that non-reactive CHR propagation rules do not require the maintenance of a propagation history. Instead, cheap constraint identifier comparisons can be used. Furthermore, these comparisons can be moved early in the generated nested loops, thus pruning the search space of possible partner constraints. We formally proved the correctness of the optimization for non-reactive rules with respect to CHR's refined operational semantics. We implemented both optimizations, and observed significant performance gains.

**Future Work** For reactive CHR rules a propagation history still has to be maintained. This includes the rules of most true constraint solvers. Most constraint solvers though, such as the archetypal LEQ handler of Example 1, have set semantics. As argued by [12] (see *Related Work* paragraph), if constraints have set

semantics, a propagation history is less compelling. Under the refined operational semantics, however, set semantics alone does not suffice to justify the elimination of propagation histories, that is without affecting a program's semantics. A stronger property called *idempotence* is required. We are currently developping an analysis to derive this property, and have already observed promising performance improvements for several programs. For certain programs, an automated confluence analysis (see e.g. [1]) would be useful, as rules that remove duplicate constraints may be moved to the front of a confluent CHR program.

*Acknowledgements* The author thanks Tom Schrijvers for his invaluable aid in the implementation of the optimizations in the K.U.Leuven CHR system. Thanks also to Bart Demoen and the anonymous referees for their useful comments.

## References

1. Frühwirth, T.: Theory and practice of Constraint Handling Rules. J. Logic Programming, Special Issue on Constraint Logic Programming **37**(1–3) (1998) 95–138
2. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. Submitted to *Journal of Theory and Practice of Logic Programming* (2008)
3. Holzbaur, C., Frühwirth, T.: A Prolog Constraint Handling Rules compiler and runtime system. Volume 14(4) of Journal of Applied Artificial Intelligence. Taylor & Francis (April 2000) 369–388
4. Schrijvers, T.: Analyses, optimizations and extensions of Constraint Handling Rules. PhD thesis, K.U.Leuven, Belgium (June 2005)
5. Van Weert, P., Schrijvers, T., Demoen, B.: K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In: CHR '05: Proc. 2nd Workshop on Constraint Handling Rules, Sitges, Spain (2005) 47–62
6. Duck, G.J.: Compilation of Constraint Handling Rules. PhD thesis, University of Melbourne, Australia (December 2005)
7. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. [13] 90–104
8. Schrijvers, T., Demoen, B.: The K.U.Leuven CHR system: Implementation and application. In Frühwirth, T., Meister, M., eds.: CHR '04, Selected Contributions, Ulm, Germany (May 2004) 8–12
9. Schrijvers, T., Stuckey, P.J., Duck, G.J.: Abstract interpretation for Constraint Handling Rules. In Barahona, P., Felty, A., eds.: PPDP '05, Lisbon, Portugal, ACM Press (July 2005) 218–229
10. Schrijvers, T., Demoen, B.: Antimonotony-based delay avoidance for CHR. Technical Report CW 385, K.U.Leuven, Dept. Comp. Sc. (July 2004)
11. Schrijvers, T., Warren, D.S.: Constraint Handling Rules and tabled execution. [13] 120–136
12. Sarna-Starosta, B., Ramakrishnan, C.: Compiling Constraint Handling Rules for efficient tabled evaluation. In Hanus, M., ed.: PADL '07: Proc. 9th Intl. Symp. Practical Aspects of Declarative Languages. Volume 4354 of LNCS., Nice, France, Springer (January 2007) 170–184
13. Demoen, B., Lifschitz, V., eds.: ICLP '04: Proc. 20th Intl. Conf. Logic Programming. In Demoen, B., Lifschitz, V., eds.: ICLP '04. Volume 3132 of LNCS., Saint-Malo, France, Springer (September 2004)

## A On Anti-Monotony-based Delay Avoidance

In [10], the following version of the **Solve** transition is proposed:

**1. Solve**$^\dagger$ $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle S +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ if $b$ is a built-in constraint and $S \subseteq \mathbb{S}$ such that $\forall c \in \mathbb{S} \setminus S : \exists V_1, V_2 : vars(c) = V_1 \cup V_2 \wedge V_1 \subseteq fixed(\mathbb{B}) \wedge$ all variables in $V_2$ appear only in arguments of $c$ that are anti-monotone in $\mathcal{P}$.

**Proposition 1.** *Using our notation, this* ***Solve***$^\dagger$ *transition is equivalent to:*

**1. Solve**$^\ddagger$ $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle S +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ if $b$ is a built-in constraint and $S \subseteq \mathbb{S}$ such that $\forall c \in \mathbb{S} \setminus S : delay\_vars_{\mathcal{P}}(c) \subseteq fixed(\mathbb{B})$.

*Proof.* Let $c \in \mathbb{S} \setminus S$, with $S$ defined as in **Solve**$^\dagger$. Then sets $V_1$ and $V_2$ exist, as defined in **Solve**$^\dagger$. By definition, $V_2 \subseteq vars(c) \backslash delay\_vars_{\mathcal{P}}(c)$, and thus $V_2 \cap delay\_vars_{\mathcal{P}}(c) = \emptyset$. Therefore, $delay\_vars_{\mathcal{P}}(c) \subseteq V_1 \subseteq fixed(\mathbb{B})$.

Conversely, assume $c \in \mathbb{S} \setminus S$, with $S$ defined as in **Solve**$^\ddagger$. Then the required sets $V_1$ and $V_2$ exist: simply take $V_1 = delay\_vars_{\mathcal{P}}(c)$ and $V_2 = vars(c) \setminus V_1$. □

In [10] the resulting semantics is shown to be correct with respect to the original refined operational semantics $\omega_r$ [7], where **Solve** is specified as:

**1. Solve**$^\star$ $\langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail_{\mathcal{P}} \langle S +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ where $b$ is a built-in constraint and $S \subseteq \mathbb{S}$ such that $vars(\mathbb{S} \setminus S) \subseteq fixed(\mathbb{B})$.

That is, all constraints with at least one non-fixed argument have to be reactivated. The original specification of the $\omega_r$ semantics therefore prohibits any form of delay avoidance for non-fixed arguments, as illustrated by this example:

*Example 5.* Consider the following CHR program:

```
c(X) ⇒ X = 2, b.
c(_), a ⇔ true.
c(_), b ⇔ true.
```

For the query '`a, c(X)`' with `X` a free logical variable, **Solve**$^\star$ specifies that the `c(X)` constraint has to be reactivated when '`X = 2`' is added to the built-in constraint solver, which leads to a final constraint store {`b#3`}. This is the only final store allowed by the original refined semantics. However, as the program is clearly anti-monotone in `c`'s argument, the **Solve**$^\ddagger$ transition might not reactivate `c`, which then leads to an incorrect final constraint store {`a#1`}.

This counterexample shows the proof in [10] must be wrong. The essential problem is that **Solve**$^\star$ specifies that constraints with non-fixed arguments have to be reactivated, even if the newly added built-in constraint does not enable any new matchings with them. This problem is not restricted to delay avoidance. It was first noted in [6, 4] in a different context:

*Example 6.* Consider the following CHR program:

```
c(X) ⇒ Y = 2, b.
c(_), a ⇔ true.
c(_), b ⇔ true.
```

For the query '`a, c(X)`' with $X$ a free logical variable the original **Solve**$^\star$ transition specifies that the `c(X)` constraint must be reactivated when '`Y = 2`' is added to the built-in constraint solver. The only final store allowed by the original refined semantics is thus {`b#3`}. However, actual CHR implementations will not reactivate the $c(X)$ constraint, as the newly added '`Y = 2`' constraint does not affect `X`, the only variable occurring in $c(X)$.

Because the original refined operational semantics is thus inconsistent with the behavior of actual (Prolog) CHR implementations, a slightly more relaxed version of the **Solve** transition was defined in [6, 4]. This is also the version of $\omega_r$ we presented in Section 2.2. The following theorem shows that our definition of **Solve'** in Section 4 is correct with respect to this relaxed $\omega_r$ semantics:

**Theorem 2.** *Let $\mathcal{P}$ be an arbitrary CHR program, and $\sigma = \langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ an arbitrary state with $b$ a built-in constraint. If $\sigma \rightarrowtail_{\mathcal{P}} \langle S +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ is a valid **Solve'** transition of $\omega_r'$, then it is a valid $\omega_r$ **Solve** transition as well.*

*Proof.* By definition of **Solve'**, $S \subseteq \mathbb{S}$, and

(1) $\forall c \in S : delay\_vars_{\mathcal{P}}(c) \not\subset fixed(\mathbb{B})$,
(2) $\forall H \subseteq \mathbb{S} : (H = K +\!\!+ R \wedge \exists \rho \in \mathcal{P} :$
$\neg appl(\rho, K, R, \mathbb{B}) \wedge appl(\rho, K, R, b \wedge \mathbb{B})) \rightarrow (S \cap H \neq \emptyset)$.

As the lowerbound of the **Solve** transition in $\omega_r$ is also exactly (2), it suffices to prove that $\forall c \in S : vars(c) \not\subset fixed(\mathbb{B})$. This is obvious given (1), as by definition $\forall c : delay\_vars_{\mathcal{P}}(c) \subseteq vars(c)$. □

The optimized semantics of [10] on the other hand remains incorrect with respect to the relaxed $\omega_r'$ semantics. The reason is that the **Solve**$^\ddagger$ transition only restricts the constraints that are not reactivated. The constraints that are reactivated, on the other hand, are not restricted:

*Example 7.* Consider the following CHR program:

```
c ⇒ X = 2, b.
c, a ⇔ true.
c, b ⇔ true.
```

For the query '`a, c`' the **Solve** transition of Fig. 2 specifies that the `c` constraint may not be reactivated when the '`X = 2`' constraint is told. This leads to the only final store allowed by the $\omega_r$ semantics of Section 2.2, namely {`b#3`}. The **Solve**$^\ddagger$ transition, however, allows the `c`'s reactivation. The resulting semantics thus may lead to an incorrect final constraint store {`a#1`}.

The final theorem show that our **Solve'** transition is indeed stronger then **Solve**$^\ddagger$, since it never reactivates more constraints:

**Theorem 3.** *Let $\mathcal{P}$ be a CHR program, and $\sigma = \langle [b|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ astate with $b$ a built-in constraint. If $\sigma \rightarrowtail_{\mathcal{P}} \langle S +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ is a valid **Solve**$^\ddagger$ transition, and $\sigma \rightarrowtail_{\mathcal{P}} \langle S' +\!\!+ \mathbb{A}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ a valid **Solve'** transition of $\omega_r'$, then $S' \subseteq S$.*

*Proof.* By definition of **Solve**$^\ddagger$: $\forall c \in \mathbb{S} \backslash S : delay\_vars_{\mathcal{P}}(c) \subseteq fixed(\mathbb{B})$, and by definition of **Solve'**: $S' \subseteq \mathbb{S} \wedge \forall c \in S' : delay\_vars_{\mathcal{P}}(c) \not\subset fixed(\mathbb{B})$. Therefore clearly $(\mathbb{S} \backslash S) \cap S' = \emptyset$, and thus $(S' \subseteq \mathbb{S} \wedge S' \cap (\mathbb{S} \backslash S) = \emptyset) \rightarrow S' \subseteq S$. □

# Modular CHR with *ask* and *tell*

François Fages, Cleyton Mario de Oliveira Rodrigues, Thierry Martinez

Contraintes Project–Team, INRIA Paris–Rocquencourt,
BP105, 78153 Le Chesnay Cedex, France.
http://contraintes.inria.fr

**Abstract.** In this paper, we introduce a modular version of the Constraint Handling Rules language CHR, called CHRat for modular CHR with *ask* and *tell*. Any constraint defined in a CHRat component can be reused both in rules and guards in another CHRat component to define new constraint solvers. Unlike previous work on modular CHR, our approach is completely general as it does not rely on an automatic derivation of conditions for checking entailment in guards, but on a programming discipline for defining both satisfiability (*tell*) and entailment (*ask*) checks by CHRat rules for each constraint. We define the operational and declarative semantics of CHRat, provide a transformation of CHRat components to flat CHR programs, and prove the preservation of the semantics. We then provide examples of the modularization of classical CHR constraint solvers and of the definition of complex constraint solvers in a modular fashion.

## 1 Introduction

The Constraint Handling Rules language CHR was introduced nearly two decades ago as a declarative language for defining constraint solvers by multiset rewriting rules with guards assuming some built-in constraints [1]. The CHR programming paradigm resolves implementing a constraint system into the declaration of guarded rewriting rules, that transform the store into a solved form allowing to decide the satisfiability. Each transformation is supposed to preserve the satisfiability of the system, and the solved form, reached when no more transformation can be applied, is unsatisfiable if it contains the constraint "false", and is operationnally satisfiable otherwise. One important, but not mandatory, property of these transformations is *confluence* which means that the solved form is always independent of the order of application of the rules, and is in fact a *normal form* for the initial constraint store [2].

Since then, CHR has evolved to a general purpose rule-based programming language [1] with some extensions such as for the handling of disjunctions [3] or for introducing types [4]. However, one main drawback of CHR as a language for defining constraint solvers, is the absence of *modularity*. Once a constraint system is defined in CHR with some built-in constraints, this constraint system cannot be reused in another CHR program taking the defined constraints as new built-in constraints. The reason for this difficulty is that a CHR program defines

a satisfiability check but not the *constraint entailment* check that is required in guards.

Previous approaches to this problem have studied conditions under which one can derive automatically an entailment check from a satisfiability check. In [5] such conditions are given based on the logical equivalence:

$$D \models C \rightarrow c \Leftrightarrow D \models (C \wedge c) \leftrightarrow C$$

In this paper, we propose a different paradigm for *modular* CHR, called CHR with *ask* and *tell*, and denoted CHRat. This paradigm is inspired by the framework of concurrent constraint programming [6, 7]. The programming discipline in CHRat for programming modular constraint solvers is to enforce, for each constraint $c$, the definition of simplification and propagation rules for the constraint tokens **ask**(c) and **entailed**(c). Solvers for *asks* and *tells* are already required for the built-in constraint system implementation [8]; the discipline we propose consists in the internalization of this requirement in the CHR solver itself. A constraint $c$ is *operationally entailed* in a constraint store containing **ask**(c) when its solved form contains the token **entailed**(c). Beside the simpagation rule c **ask**(c) $\Longrightarrow$ **entailed**(c) which will be always assumed to provide a minimalist entailment-solver, arbitrarily complex entailment checks can be programmed with rules, as opposed to event-driven imperative programming[9]. With this programming discipline, CHRat constraints can be reused both in rules and guards in other components to define new constraint solvers.

In the next section, we illustrate this approach with a simple example. Then we define the syntax and declarative semantics of CHRat. Section 4 describes the transformation of CHRat programs into flat CHR programs and proves its correctness. Section 5 provides examples of the modularization of classical CHR constraint solvers and of the definition of complex constraint solvers in a modular fashion. Finally we conclude with a discussion on the simplicity and expressiveness of this approach and its current limitation to non-quantified constraints.

## 2 Introductory Example

### 2.1 CHRat Components for leq/2 and min/3

We begin with the pedagogical CHR constraint solver for ordering relations. This solver defines the CHR constraint leq/2. The first task is to define, as usual, the satisfiability solver associated to this constraint: this is done by the following four rules. The first three rules translate the axioms for ordering relations, and the rule redundant gives set semantics to the constraint leq/2.

File leq_solver.cat

```
component leq_solver.
export leq/2.
reflexive     @ leq(X,X) <=> true.
antisymmetric @ leq(X,Y), leq(Y,X) <=> X = Y.
transitive    @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
redundant     @ leq(X,Y) \ leq(X,Y) <=> true.
```

There is a second task for defining a constraint solver in CHRat: the definition of rules for checking the entailment of leq(X, Y) constraint. These rules have to rewrite the constraint token **ask**(leq(X, Y)) into the constraint token **entailed**(leq(X, Y)). The rule leq(X,Y) \ **ask**(leq(X,Y)) ⟺**entailed**(leq(X,Y)) is always assumed and provides a minimalist entailment-solver for free. In this simple example, since checking leq(X, Y) for $X \neq Y$ is directly observable in the store, there is only a single rule to add for the reflexivity.

```
reflexiveAsk @ ask(leq(X,X)) ⟺ entailed(leq(X,X)).
```

The satisfiability solver and the entailment solver together define a CHRat component for the CHR-constraint leq(X, Y). Our implementation of CHRat relies on a simple atom-based component separation mechanism: there is a component by file; exported CHR-constraints are prefixed with the name of the component; and the choice for the prefixes of internal CHR-constraints is done so as to avoid collisions.

Such a component can then be used to define new constraint solvers using the leq(X, Y) constraint both in rules and guards. For instance, a component for the minimum constraint min(X,Y,Z), stating that Z is the minimum value among X and Y, can be defined in CHRat as follows:

File min_solver.cat

```
component min_solver.
import leq/2 from leq_solver.
export min/3.
minLeft     @ min(X,Y,Z) ⟺ leq(X,Y) | Z=X.
minRight    @ min(X,Y,Z) ⟺ leq(Y,X) | Z=Y.
minGen      @ min(X,Y,Z) ⟹ leq(Z,X), leq(Z,Y).

minAskLeft  @ ask(min(X, Y, X)) ⟺ leq(X, Y) |
                entailed(min(X, Y, X)).
minAskRight @ ask(min(X, Y, Y)) ⟺ leq(Y, X) |
                entailed(min(X, Y, Y)).
```

The three first rules describe the satisfiability check for min(X,Y,Z). The relevant rules to be discussed are the minAskLeft and minAskRight: it is worth noticing that the entailment of min(X,Y,Z) can be stated if, and only if, Z is already known to be equal to X or Y.

### 2.2 Transformation to a Flat CHR Program

The guards in CHR rules are restricted to built-in constraints [1]. In order to translate CHRat programs into CHR programs, we proceed with a program transformation which removes all the user defined constraints from the guard. This transformation also renames the constraints **ask**( constraint ) to ask_constraint and **entailed**( constraint ) to entailed_constraint . The resulting CHR program for the min(X, Y, Z) CHRat solver is the following:

```
min-auto-ask      @ min(X,Y,Z) \ ask_min(X,Y,Z) ==> entailed_min(X,Y,Z).
minLeft-ask       @ min(X,Y,Z) ==> ask_leq(X,Y).
minLeft-fire      @ entailed_leq(X,Y), min(X,Y,Z) <==> Z=X.
minRight-ask      @ min(X,Y,Z) ==> ask_leq(Y,X).
minRight-fire     @ entailed_leq(Y,X), min(X,Y,Z) <==> Z=Y.
minGen            @ min(X,Y,Z) ==> leq(Z,X), leq(Z,Y).
minAskLeft-ask    @ ask_min(X,Y,X) ==> ask_leq(X,Y).
minAskLeft-fire   @ entailed_leq(X,Y), ask_min(X,Y,X) <==>
                        entailed_min(X,Y,X).
minAskRight-ask   @ ask_min(X,Y,Y) ==> ask_leq(Y,X).
minAskRight-fire  @ entailed_leq(Y,X), ask_min(X,Y,Y) <==>
                        entailed_min(X,Y,Y).
```

It is worth noting that the transformed program can be executed with any regular CHR implementation [10, 11].

## 3 Syntax and Semantics of CHRat Components

Let $V$ be a countable set of variables. Let $\mathrm{fv}(e)$ denotes the set of free variables of a formula $e$.

### 3.1 Syntax

**Definition 1.** *A* built-in constraint system *is a pair* $(\mathcal{C}, \vdash_{\mathcal{C}})$*, where:*

- $\mathcal{C}$ *is a set of formulas over the variables* $V$*, closed by logical operators and quantifiers;*
- $\Vdash_{\mathcal{C}} \subseteq \mathcal{C}^2$ *defines the non-logical axioms of the constraint system;*
- $\vdash_{\mathcal{C}}$ *is the least subset of* $\mathcal{C}^2$ *containing* $\Vdash_{\mathcal{C}}$ *and closed by the logical rules.*

Let $(\mathcal{C}, \vdash_{\mathcal{C}})$ be a built-in constraint system over a domain $\mathcal{D}$ with variables $V$, assumed to contain the standard axiom schemas for equality.

Let $\mathcal{T}$ be a set of constraint *tokens* of the form $c(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n \in \mathcal{D}$ and disjoint from $\mathcal{C}$. We suppose that for any $t \in \mathcal{T}$, $\mathrm{ask}(t) \notin \mathcal{T}$ and $\mathrm{entailed}(t) \notin \mathcal{T}$, and we define

$$\mathcal{T}_a \doteq \{\mathrm{ask}(t) \mid t \in \mathcal{T}\}$$
$$\mathcal{T}_e \doteq \{\mathrm{entailed}(t) \mid t \in \mathcal{T}\}$$

$\mathcal{T}$, $\mathcal{T}_a$, $\mathcal{T}_e$ and $\mathcal{C}$ are thus pairwise disjoint. Let:

$$\mathcal{T}^\bullet \doteq \mathcal{T} \uplus \mathcal{T}_a \uplus \mathcal{T}_e$$

where $\uplus$ denotes disjoint set union.

$\mathrm{ask}^*(\cdot)$ and $\mathrm{entailed}^*(\cdot)$ are the homomorphic extensions of $\mathrm{ask}(\cdot)$ and $\mathrm{entailed}(\cdot)$ respectively to functions from multisets to multisets.

**Definition 2.** *A* CHRat rule *is of one of the three forms that follow:*

**Simplification** *rule-name* @ $H \Leftrightarrow G \mid B$.
**Propagation** *rule-name* @ $H \Rightarrow G \mid B$.
**Simpagation** *rule-name* @ $H \setminus H' \Leftrightarrow G \mid B$.

*where*

- *rule-name is an optional name for the rule;*
- *the* heads $H$ *and* $H'$ *are non-empty multisets of elements of* $\mathcal{T} \uplus \mathcal{T}_a \uplus \mathcal{C}$*;*
- *the* guard $G$ *is a multiset of elements of* $\mathcal{T} \uplus \mathcal{C}$*;*
- *the* body $B$ *is a multiset of elements of* $\mathcal{T} \uplus \mathcal{T}_e \uplus \mathcal{C}$.

In guards, the built-in constraints will be distinguished from the user-defined constraints. For a guard $C$, we write $C_{\text{built-in}} = C \cap \mathcal{C}$ and $C_{\text{CHR}} = C \cap \mathcal{T}$.

**Definition 3.** *A* CHRat program *is a tuple* $(\{r_1, \ldots, r_n\}, \Sigma)$ *where* $r_1, \ldots, r_n$ *are* CHRat *rules, and* $\Sigma$ *is the signature of* $\mathcal{T}$*, with the following side condition: for every rule, all variables which appear in the* CHR*-constraint part of the guard* $C_{\text{CHR}}$*, also appear in the head or in the built-in constraints of the guard.*

*Remark 1.* It is worth noticing that the restriction for guards in CHRat only concerns the CHR-constraint part. In particular, since a CHR program has no CHR-constraint in its guards, every CHR program is a valid CHRat program.

In principle, CHRat programs for *ask* should satisfy some further properties. Putting an ask(·) token should indeed never lead to a failure, and an ask solver should restrict its interaction with the store such that, as far as other components are concerned, only consumption of ask(·) tokens and addition of entailed(·) tokens can be observed, in particular rules for *ask* should not add *tell* constraints to the store. However, the formal semantics described in the following sections will not assume these further restrictions.

As usual, and without loss of generality, we will focus on slightly generalized simpagation rules where one of the heads can be empty. Simplification rules and propagation rules will then be mapped to simpagation rules, by assuming that left heads are empty in translations of simplification rules, and that right heads are empty in translations of propagation rules.

### 3.2 Operational Semantics

As usual, the operational semantics of CHRat is defined as a transition system between states, called configurations, defined as for CHR [1] by:

**Definition 4.** *A* configuration *is a tuple* $\langle F, E, D \rangle_{\mathcal{V}}$ *where:*

- *the query* $F$ *is a multiset of elements from* $\mathcal{C} \uplus \mathcal{T}^\bullet$*;*
- *the* CHRat *constraint store* $E$ *is a multiset of elements from* $\mathcal{T}^\bullet$*;*
- *the built-in store* $D$ *is an element of* $\mathcal{C}$*;*
- $\mathcal{V} \subset V$ *is the set of variables of the initial query.*

Let $\mathfrak{C}$ denotes the set of all configurations.

**Definition 5.** *We distinguish some relevant configurations:*

- *an* initial *configuration is of the form* $\langle F, \emptyset, \text{true}\rangle_{\mathcal{V}}$ *where* $\mathcal{V} = \text{fv}\,(F)$*;*
- *a* failed *configuration is of the form* $\langle F, E, D\rangle_{\mathcal{V}}$ *where* $D \vdash_{\mathcal{C}} \text{false}$*;*
- *a* successful *configuration is of the form* $\langle \emptyset, E, D\rangle_{\mathcal{V}}$ *where* $D \not\vdash_{\mathcal{C}} \text{false}$*.*

The set of variables of the initial query is written in the configuration to keep these variables free when we consider the logical meaning of the configuration:

**Definition 6.** *The* logical meaning *of a configuration:*

$$\langle F, E, D\rangle_{\mathcal{V}}$$

*is:*

$$\exists \boldsymbol{y}(F \wedge E \wedge D)$$

*where* $\boldsymbol{y}$ *enumerates* $\text{fv}\,(F, E, D) \setminus \mathcal{V}$*.*

**Definition 7.** *Let* $P$ *be a* CHRat *program. The* transition relation $\mapsto \subseteq \mathfrak{C}^2$ *is the least binary relation closed by the following induction rules:*

**Solve**

$$\frac{c \in \mathcal{C}}{\langle \{c\} \uplus F, E, D\rangle_{\mathcal{V}} \mapsto \langle F, E, c \wedge D\rangle_{\mathcal{V}}}$$

**Introduce**

$$\frac{t \in \mathcal{T}^{\bullet}}{\langle \{t\} \uplus F, E, D\rangle_{\mathcal{V}} \mapsto \langle F, \{t\} \uplus E, D\rangle_{\mathcal{V}}}$$

**Trivial Entailment**

$$\frac{t \in \mathcal{T}}{\langle F, \{\text{ask}(t), t\} \uplus E, D\rangle_{\mathcal{V}} \mapsto \langle \{\text{entailed}(t)\} \uplus F, \{t\} \uplus E, D\rangle_{\mathcal{V}}}$$

**Ask**

$$\frac{(H \setminus H' \Leftrightarrow C_{\text{built-in}}, C_{\text{CHR}} \mid B.)\,\sigma \in P \qquad D \vdash_{\mathcal{C}} C_{\text{built-in}}}{\langle F, H \uplus H' \uplus E, D\rangle_{\mathcal{V}} \mapsto \langle \text{ask}^*(C_{\text{CHR}}) \uplus F, H \uplus H' \uplus E, D\rangle_{\mathcal{V}}}$$

**Fire**

$$\frac{(H \setminus H' \Leftrightarrow C_{\text{built-in}}, C_{\text{CHR}} \mid B.)\,\sigma \in P \qquad D \vdash_{\mathcal{C}} C_{\text{built-in}}}{\langle F, H \uplus H' \uplus \text{entailed}^*(C_{\text{CHR}}) \uplus E, D\rangle_{\mathcal{V}} \mapsto \langle B \uplus F, H \uplus E, D\rangle_{\mathcal{V}}}$$

*where* $\sigma$ *denotes some variable substitution: the support of* $\sigma$ *consists of the free variables appearing in the rule which* $\sigma$ *is applied to; in the* ***Ask*** *rule, variables which do not appear in* $H$*,* $H'$*,* $C_{\text{built-in}}$ *have to be mapped to fresh variables; in the* ***Fire*** *rule, variables which do not appear in* $H$*,* $H'$*,* $C_{\text{built-in}}$*,* $C_{\text{CHR}}$ *have to be mapped to fresh variables.*

Whereas a CHR rule is reduced in only one step, CHRat reduces it in two steps: first, if the heads and builtin guards match, ask solvers are awoken with ask(·) tokens (**Ask** rule); then, when all ask solvers have answered positively to the guard with entailed(·) tokens, the body of the rule is fired (**Fire** rule). Unlike deep guards [12], asks are thus checked in CHRat in the same constraint store as tells.

**Definition 8.** *A* computation of a goal $G$ *is a sequence* $S_0, S_1, \ldots$ *of configurations with* $S_i \mapsto S_{i+1}$*, beginning with* $S_0 = \langle G, \emptyset, \mathrm{true}\rangle_{\mathcal{V}}$ *and ending in a final configuration or diverging. A finite computation is* successful *if the final configuration is successful. It is* failed *otherwise. The logical meaning of the final configuration of a finite computation is called the* answer *of the computation.*

### 3.3 Declarative Semantics

CHR programs enjoy a logical semantics that is better suited than the operational semantics to reason about programs and establish program equivalence for instance. In this section, we show that this logical reading of the rules applies as well to CHRat programs.

Let $\mathcal{C}^\bullet$ be the closure of $\mathcal{C} \uplus \mathcal{T}^\bullet$ by logical operators and quantifiers, and let $\vdash_{\mathcal{C}^\bullet}$ be the logical extension of $\vdash_{\mathcal{C}}$ to $\mathcal{T}^\bullet$ with equality and no other non-logical axiom. More precisely, $\vdash_{\mathcal{C}^\bullet}$ is the closure of $\Vdash_{\mathcal{C}^\bullet}$ by logical rules with:

$$\Vdash_{\mathcal{C}^\bullet} \doteq \Vdash_{\mathcal{C}} \uplus \left\{ \begin{array}{r} (c(x_1,\ldots,x_n), c(x'_1,\ldots,x'_n)) \in (\mathcal{T}^\bullet)^2 \\ \mid \vdash_{\mathcal{C}} x_1 = x'_1 \wedge \cdots \wedge x_n = x'_n \end{array} \right\}$$

**Definition 9.** *Let:*

$$(\cdot)^\ddagger : \mathsf{CHRat} \to \mathcal{C}^\bullet$$

*be defined for* CHRat *rules as follows:*

$$\begin{aligned} (\mathit{rule} \ @\ H \setminus H' \Leftrightarrow C_{\text{built-in}}, C_{\text{CHR}} \mid B.)^\ddagger \doteq \\ \forall \boldsymbol{y}(C_{\text{built-in}} \to \overline{H} \wedge \overline{H'} \to \overline{\mathrm{ask}^*(C_{\text{CHR}})}) \\ \wedge \forall \boldsymbol{y}(C_{\text{built-in}} \to (\overline{H} \wedge \overline{H'} \wedge \overline{\mathrm{entailed}^*(C_{\text{CHR}})} \leftrightarrow \exists \boldsymbol{y}'(\overline{H} \wedge \overline{B}))) \end{aligned}$$

*where:*

- $\boldsymbol{y}$ *enumerates the variables occurring in the head and the guard, and* $\boldsymbol{y}'$ *enumerates the other variables occurring in the body (without occurring neither in the head nor in the guard);*
- *for each multiset of constraints* $S = \{c_1, \ldots, c_n\}$*,* $\overline{S}$ *denotes the constraint* $c_1 \wedge \cdots \wedge c_n$*;*

*The* declarative semantics *of a program* $P \doteq (\{r_1, \ldots, r_n\}, \Sigma)$ *is:*

$$(P)^\ddagger \doteq \left( \bigwedge_{1 \le i \le n} (r_i)^\ddagger \right) \wedge \left( \bigwedge_{(f/k) \in \Sigma} \begin{array}{r} \forall \boldsymbol{x}(f(x_1,\ldots,x_k) \to \\ (\mathrm{ask}(f(x_1,\ldots,x_k)) \leftrightarrow \mathrm{entailed}(f(x_1,\ldots,x_k)))) \end{array} \right)$$

*Remark 2.* Let $(\cdot)^\dagger$ denote the usual CHR declarative semantics. For all CHR program $P$, we have $\vdash_{\mathcal{C}^\bullet} (P)^\ddagger \leftrightarrow (P)^\dagger$: CHR is thus a proper sublanguage of CHRat and CHRat, both syntactically and semantically.

The fundamental link between the operational and the declarative semantics is stated by:

**Lemma 1.** *Let $P$ be a* CHRat *program and $S \mapsto S'$ be a transition. Let $C$ and $C'$ denote the logical reading of $S$ and $S'$ respectively. We have:*

$$(P)^{\ddagger} \vdash_{\mathcal{C}\bullet} \forall \boldsymbol{x}(C \leftrightarrow C')$$

*where $\boldsymbol{x}$ enumerates $fv(C) \cup fv(C')$.*

*Proof.* Case analysis over the kind of the transition $S \mapsto S'$:

- immediate for **Solve** and **Introduce**; **Trivial Entailment** derives from $(P)^{\ddagger} \vdash_{\mathcal{C}\bullet} c \rightarrow (\mathrm{ask}(c) \leftrightarrow \mathrm{entailed}(c))$ for all $c \in \mathcal{T}$;
- **Ask** and **Fire** derive from the logical translation of the CHRat rule which they are applied to.

This result is the direct translation for CHRat of Fruhwirth's soundness and completeness results for CHR. As such, this lemma entails the soundness and completeness of the operational semantics with respect to the declarative semantics:

**Theorem 1 (Soundness).** *Let $P$ be a* CHRat *program and $G$ be a goal. If $G$ has a computation with answer $C$ then:*

$$(P)^{\ddagger} \vdash_{\mathcal{C}\bullet} \forall \boldsymbol{x}(C \leftrightarrow G)$$

*where $\boldsymbol{x}$ enumerates free variables of $C$ and $G$.*

**Theorem 2 (Completeness).** *Let $P$ be a* CHRat *program and $G$ be a goal with at least one finite computation. For all conjunctions of constraints $C$ such that $(P)^{\ddagger} \vdash_{\mathcal{C}\bullet} \forall \boldsymbol{x}(C \leftrightarrow G)$, there exists a computation of answer $C'$ from $G$ such that:*

$$(P)^{\ddagger} \vdash_{\mathcal{C}\bullet} \forall \boldsymbol{x}(C \leftrightarrow C')$$

*where $\boldsymbol{x}$ enumerates free variables of $C$ and $C'$.*

## 4 Program Transformation of CHRat to CHR

**Definition 10.** *Let $[\![\cdot]\!] : \mathsf{CHRat} \rightarrow \mathsf{CHR}$ be defined for every* CHRat *rule by $[\![\cdot]\!]$ as follows:*

$$[\![\mathit{rule} \ @ \ H \ \setminus \ H' \ \Leftrightarrow \ C_{\text{built-in}}, C_{\text{CHR}} \mid B.]\!] \doteq \begin{cases} \mathit{rule\text{-}ask} \ @ \ H, H' \ \Rightarrow \ C_{\text{built-in}} \mid \mathrm{ask}^*(C_{\text{CHR}}). \\ \mathit{rule\text{-}fire} \ @ \ H \ \setminus \ H', \mathrm{entailed}^*(C_{\text{CHR}}) \ \Leftrightarrow \ C_{\text{built-in}} \mid B. \end{cases} \quad (1)$$

*Transformations for simplification and propagation rules follow from 1 by immediate specialization. The image of a whole* CHRat *program* $(R, \Sigma)$ *by* $[\![\cdot]\!]$ *is the concatenation of images of the individual rules, with the propagation rules:*

$$f(x_1, \ldots, x_k) \;\Rightarrow\; \text{entailed}(f(x_1, \ldots, x_k)).$$

*implicitly added for each constraint declaration* $(f/k) \in \Sigma$*, if such a rule was not already written by the user in* $R$*.*

To take benefits of first functor symbol indexing, instead of using compound terms **ask** (...) or **entailed** (...) in the implementation, we rather prefix the constraint symbol in the generated CHR code: constraint (x, y, z) becomes ask_constraint (x, y, z) and entailed_constraint (x, y, z). We could otherwise relying upon automatic program transformations to make this optimization [13].

It is worth noticing that the first and the second CHR rules produced for each CHRat rule respectively follow the right and the left operands of $\wedge$ in the declarative semantics of this rule. That leads to the following result which states the soundness of the transformation:

**Theorem 3.** *For all* CHRat *program* $P$*, we have:*

$$\vdash_{\mathcal{C}\bullet} (P)^{\ddagger} \leftrightarrow ([\![P]\!])^{\dagger}$$

*where* $(P)^{\ddagger}$ *is the* CHRat *declarative semantics of* $P$ *(see definition 9) and* $(\cdot)^{\dagger}$ *denotes the usual* CHR *declarative semantics (see remark 2)*

*Proof.* Let $P = (\{r_1, \ldots, r_n\}, \Sigma)$. According to the definitions:

$$\vdash_{\mathcal{C}\bullet} (P)^{\ddagger} \leftrightarrow \left( \bigwedge_{1 \le i \le n} (r_i)^{\ddagger} \right) \wedge e \text{ and } \vdash_{\mathcal{C}\bullet} ([\![P]\!])^{\dagger} \leftrightarrow \left( \bigwedge_{1 \le i \le n} ([\![r_i]\!])^{\dagger} \right) \wedge e$$

where:

$$e \doteq \left( \bigwedge_{(f/k) \in \Sigma} \begin{array}{l} \forall \boldsymbol{x}(f(x_1, \ldots, x_k) \rightarrow \\ \quad (\text{ask}(f(x_1, \ldots, x_k)) \leftrightarrow \text{entailed}(f(x_1, \ldots, x_k)))) \end{array} \right)$$

Then it suffices to show that, for all $1 \le i \le n$, $\vdash_{\mathcal{C}\bullet} (r_i)^{\ddagger} \leftrightarrow ([\![r_i]\!])^{\dagger}$. Let $r_i$ be the simpagation rule:

$$H \;\setminus\; H' \;\Leftrightarrow\; C_{\text{built-in}}, C_{\text{CHR}} \mid B.$$

then:

$$\begin{aligned} [\![r_i]\!] = \{ & (H, \text{entailed}^*(C_{\text{CHR}}) \;\setminus\; H' \;\Leftrightarrow\; C_{\text{built-in}} \mid B.), \\ & (H, H' \;\Rightarrow\; C_{\text{built-in}} \mid \text{ask}^*(C_{\text{CHR}}).) \} \end{aligned}$$

then we have:

$$\begin{aligned} ([\![r_i]\!])^{\dagger} = {} & \forall \boldsymbol{x} \left( C_{\text{built-in}} \rightarrow \overline{H} \wedge \overline{H'} \rightarrow \overline{\text{ask}^*(C_{\text{CHR}})} \right) \\ & \wedge \forall \boldsymbol{x} \left( C_{\text{built-in}} \rightarrow \left( \overline{H} \wedge \overline{H'} \wedge \overline{\text{entailed}^*(C_{\text{CHR}})} \leftrightarrow \overline{H} \wedge \exists \boldsymbol{z}(\overline{B}) \right) \right) \end{aligned}$$

where $\boldsymbol{x}$ enumerates variables of $C_{\text{built-in}}$ and $H$, and $\boldsymbol{z}$ enumerates variables of $B$. Since $P$ is a CHRat program, variables in $\text{ask}^*(C_{\text{CHR}})$ and $\text{entailed}^*(C_{\text{CHR}})$ are in $\boldsymbol{x}$. Thus $\vdash_{\mathcal{C}\bullet} (r_i)^\ddagger \leftrightarrow ([\![r_i]\!])^\dagger$.

## 5 Examples

### 5.1 Union-Find Constraint Component

The union-find (or disjoint set union) algorithm [14] has been implemented in CHR with its best-known algorithmic complexity [15]. This positive result is remarkable because logic programming paradigm has been known to be ill-suited to such implementations [16], and because the algorithm given in [15] indeed benefits from the non-monotonic evolution of the store of the operational semantics.

The union-find algorithm maintains a partition of a universe, such that each equivalence class has a *representative* element. Three operations define this data structure:

- make(X) adds the element X to the universe, initially in an equivalence class reduced to the singleton {X}.
- find(X) returns the representative of the equivalence class of X.
- union(X,Y) joins the equivalence classes of X and Y (possibly changing the representative).

**Naive Implementation** The naive implementation, which [15] begins with, relies on the classical representation of equivalence classes by rooted trees. Roots are representative elements, they are marked as such with the CHR-constraint root(X). Tree branches are marked with A ⇝ B, where A is the child and B the parent node.

File naive_union_find_solver.cat

```
component naive_union_find_solver.
export make/1, ≃/2.
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).
```

This implementation supposes that its entry-points make and union are used with constant arguments only, and that the first argument of find is always a constant.

In CHRat, one needs to add to this implementation the ability to check if two elements A and B are in the same equivalence class: we denote such a constraint

A ≃ B, where A and B are supposed to be constants. Telling this constraint just yields to the union of the two equivalence classes:

```
tellSame @ A ≃ B ⟹ union(A, B).
```

A way to provide a naive implementation for ask is to follow tree branches until possibly finding a common ancestor for A and B.

```
askEq    @ ask(A ≃ A) ⟺ entailed(A ≃ A).
askLeft  @ A ⇝ C \ ask(A ≃ B) ⟺ C ≃ B | entailed(A ≃ B).
askRight @ B ⇝ C \ ask(A ≃ B) ⟺ A ≃ C | entailed(A ≃ B).
```

The computation required to check the constraint entailment is done with the use of recursion in the definition of the guard $A = B$.

**Optimized Implementation** The second implementation proposed in [15] implements both *path-compression* and *union-by-rank* optimizations.

File union_find_solver.cat

```
component union_find.
export make/1, ≃/2.
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(M+1, N), root(A, N1).
linkRight @ link(B, A), root(A, N), root(B, M) ⟺ N ≥ M |
              B ⇝ A, N1 is max(M+1, N), root(A, N1).
```

An optimized check for common equivalence class can rely on find to efficiently get the representatives and then compare them. check(A, B, X, Y) represents the knowledge that the equivalence class representatives of A and B are the roots X and Y respectively. When X and Y are known to be equal, **entailed**(A ≃ B) is put to the store (checkEq).

```
askEq     @ ask(A ≃ B) ⟺
                find(A, X), find(B, Y), check(A, B, X, Y).
checkEq   @ root(X) \ check(A, B, X, X) ⟺ entailed(A ≃ B).
```

These two rules are not enough to define a complete entailment-solver due to the non-monotonous nature of the changes applied to the tree structure. Indeed, roots found for A and B can be invalidated by subsequent calls to union, which may transform these roots into child nodes. When a former root becomes a child node, the following two rules put find once again to get the new root.

```
checkLeft  @ X ⇝ C \ check(A, B, X, Y) ⟺
                find(A, Z), check(A, B, Z, Y).
checkRight @ Y ⇝ C \ check(A, B, X, Y) ⟺
                find(B, Z), check(A, B, X, Z).
```

These rules define complete solvers for satisfaction and entailment checking for the $\simeq$ constraint.

### 5.2 Rational Tree Equality Constraint Component

Let us now consider rational terms, i.e. rooted, ordered, unranked, labelled, possibly infinite trees, with a finite number of structurally distinct sub-trees [17]. Nodes are supposed to belong to the universe considered by the union-find solver; two nodes belonging to the same equivalence class are supposed to be structurally equal. Each node X has a signature F/N, where F is the label of X and N its arity: the associated constraint is denoted fun(X, F, N). The constraint arg(X, I, Y), for each I between 1 and N, states that the Ith subtree of X is (structurally equal to) Y. These constraints have just to be compatible between elements of the same equivalence class:

File rational_tree_solver.cat

```
component rational_tree_solver.
import ≃/2 from union_find_solver.
export fun/3, arg/3, ∼/2.
eqFun @ fun(X0, F0, N0) \ fun(X1, F1, N1) ⟺ X0 ≃ X1 |
           F0 = F1, N0 = N1.
eqArg @ arg(X0, N, Y0) \ arg(X1, N, Y1) ⟺ X0 ≃ X1 |
           Y0 ≃ Y1.
```

Telling that two trees are structurally equal, denoted $X \sim Y$, can be reduced to the union of the two equivalence classes.

```
eqProp @ X ∼ Y ⟺ X ≃ Y.
```

The computation associated to asking $A \sim B$ requires a coinductive derivation of structural comparisons to break infinite loops. That is done here by memoization. Each time a $A \sim B$ is asked, a new fresh variable M is introduced:

```
askEq @ ask(A ∼ B) ⟺ checkTree(M, A, B).
```

This variable marks the checking(M, A, B) tokens, signaling that A can be assumed to be equal to B since this check is already in progress.

```
checkTree(M, A, B) ⟺ eqTree(M, A, B) | entailed(A ∼ B).

ask(eqTree(M, A, B)) ⟺
   checking(M, A, B), fun(A, FA, NA), fun(B, FB, NB),
   checkTreeAux(M, A, B, FA, NA, FB, NB).
```

checkTreeAux firstly checks that signatures of A and B are equal, then compares arguments.

```
checkTreeAux(M, A, B, F, N, F, N) ⟺
   askArgs(M, A, B, 1, N), collectArgs(M, A, B, 1, N).
```

askArgs adds every askArg token corresponding to each pair of point-wise subtrees of A and B. askArg answers entailedArg if they match. collectArg ensures every entailedArg token have been put before concluding about the entailment of eqTree(M, A, B). It is very close to the definition of an ask solver, but the considered guard deals with a variable number of tokens equals to the arity of A and B.

```
askArgs(M, A, B, I, N) ⟺ I ≤ N |
   arg(A, I, AI), arg(B, I, BI),
   askArg(M, A, B, I, AI, BI),
   J is I + 1, askArgs(M, A, B, J, N).
askArgs(M, A, B, I, N) ⟺ true.
collectArgs(M, A, B, I, N), entailedArg(M, A, B, I) ⟺
   J is I + 1, collectArgs(M, A, B, J, N).
collectArgs(M, A, B, I, N) ⟺ I > N |
   entailed(eqTree(M, A, B)).
```

askArg firstly checks if the equality is memoized. Otherwise, the eqTree guard is recursively asked.

```
checking(M, AI, BI) \ askArg(M, A, B, I, AI, BI) ⟺
   entailedArg(M, A, B, I).
askArg(M, A, B, I, AI, BI) ⟺ eqTree(M, AI, BI) |
   entailedArg(M, A, B, I).
```

It is worth noticing that some garbage collection tasks are missing in this example: memoization tokens checking(M,A,B) are never removed from the store, and disentailment cases are not cleaned up.

## 6 Conclusion and perspectives

We have shown that by letting the programmer define in CHRat not only satisfiability checks but also entailment checks for constraints, CHRat becomes fully modular, i.e. constraints defined in one component can be reused in rules and guards in other components without restriction. Furthermore, this programming discipline is not too demanding for the programmer, as for any constraint $c$, the CHRat rule c \ **ask**(c) ⟺**entailed**(c) constitutes a default rule for checking entailment by simple store inspection. In the general case, however, CHRat rules for **ask**(c) perform arbitrary complex simpagations, which lead either to the constraint token **entailed**(c), or to another store waiting for more information.

The operational and declarative semantics of CHRat have been defined and the program transformation from CHRat to CHR which is at the basis of our compiler has been proved to implement the formal semantics of CHRat. It is worth noticing that the described transformation uniform and compatible with other orthogonal approaches related to modularity, like methodologies to make several constraint solvers collaborate [18].

We have also shown that the classical examples of constraint solvers defined in CHR could easily be modularized in CHRat and reused for building complex constraint solvers.

As for future work, the CHRat scheme can be improved in several ways. Variables in a CHRat guard have to appear either in the head or in the built-in constraint part of the guard. One way to allow existentially quantified variables guards without this restriction is to explicitly stratify guards as proposed in [9].

On the programming discipline side, ask-solvers should never lead to a failure and should not interfere with the logical interpretation of exported constraints present in the CHR store. The union-find example is a typical case where the ask-solver does change the logical meaning of the store by path-compressions and meanwhile keeps the interpretation of the exported constraint $\simeq$ unchanged. Formalizing sanity conditions for ask-solvers will be a step towards establishing the link between ask-solvers and logical entailment in the semantics.

The transformation of CHRat programs into regular CHR programs make the implementation directly benefit from optimized CHR implementations. While the efficient management of **ask** and **entailed** is left to the underlying CHR implementation, garbage collection, caching and memoization for checking entailment are left to the CHRat programmer. Good strategies for garbage collection still need to be investigated as shown in the rational tree solver.

Finally, the issue of separate compilation has not been discussed here but is a natural subject for future work in this framework.

**Acknowledgment**

We are grateful to Rishi Kumar for his preliminary work along these lines at INRIA, and to Jacques Robin for his recent impulse for this work.

## References

1. Frühwirth, T.W.: Theory and practice of constraint handling rules. J. Log. Program. **37**(1-3) (1998) 95–138
2. Abdennadher, S.: Operational semantics and confluence of constraint propagation rules. In: Proceedings of CP'1997, 3rd International Conference on Principles and Practice of Constraint Programming. Volume 1330 of Lecture Notes in Computer Science., Linz, Springer-Verlag (1997) 252–266
3. Abdennadher, S., Schütz, H.: CHRv: A flexible query language. In: FQAS '98: Proceedings of the Third International Conference on Flexible Query Answering Systems, London, UK, Springer-Verlag (1998) 1–14
4. Coquery, E., Fages, F.: A type system for CHR. In: Recent Advances in Constaints, revised selected papers from CSCLP'05. Number 3978 in Lecture Notes in Artificial Intelligence, Springer-Verlag (2006) 100–117
5. Schrijvers, T., Demoen, B., Duck, G., Stuckey, P., Fruhwirth, T.: Automatic implication checking for CHR constraint solvers. Electronic Notes in Theoretical Computer Science **147** (January 2006) 93–111
6. Saraswat, V.A.: Concurrent constraint programming. ACM Doctoral Dissertation Awards. MIT Press (1993)

7. Hentenryck, P.V., Saraswat, V.A., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(FD). Journal of Logic Programming **37**(1-3) (1998) 139–164
8. Duck, G.J., Stuckey, P.J., de la Banda, M.G., Holzbaur, C.: Extending arbitrary solvers with constraint handling rules. In: PPDP '03, Uppsala, Sweden, ACM Press (2003) 79–90
9. Duck, G.J., de la Banda, M.J.G., Stuckey, P.J.: Compiling ask constraints. In: ICLP. (2004) 105–119
10. Sneyers, J., Weert, P.V., Koninck, L.D., Demoen, B., Schrijvers, T.: The K.U.Leuven CHR system (2008)
11. Kaeser, M.: WebCHR (2007) `http://chr.informatik.uni-ulm.de/~webchr/`.
12. Schulte, C.: Programming deep concurrent constraint combinators. In Pontelli, E., Costa, V.S., eds.: Practical Aspects of Declarative Languages. PADL 2000. Volume 1753 of Lecture Notes in Computer Science., Boston, MA, USA, Springer-Verlag (2000) 215–229
13. Sarna-Starosta, B., Schrijvers, T.: Indexing techniques for CHR based on program transformation. Technical report, CW 500, K.U.Leuven, Department of Computer Science (2007)
14. Tarjan, R.E., van Leeuwen, J.: Worst-case analysis of set union algorithms. In: J. ACM. (1984)
15. Schrijvers, T., Frühwirth, T.W.: Analysing the CHR implementation of unionfind. In: 19th Workshop on (Constraint) Logic Programming. (2005)
16. Ganzinger, H.: A new metacomplexity theorem for bottom-up logic programs. In: Proceedings of International Joint Conference on Automated Reasoning. (2001)
17. Courcelle, B.: Fundamental properties of infinite trees. Theoretical Computer Science **25** (1983) 95–169
18. Abdennadher, S., Fruhwirth, T.: Integration and optimization of rule-based constraint solvers (2003)

# Default Reasoning in CHR$^\vee$

Marcos Aurélio[1,2], François Fages[2], Jacques Robin[1]

[1] Universidade Federal de Pernambuco, Recife, Brazil
[2] INRIA, Rocquencourt, France

**Abstract.** CHR$^\vee$ has emerged as a versatile knowledge representation language, usable for an unparalleled variety of automated reasoning tasks: constraint solving, optimization, classification, subsumption, classical deduction, abduction, truth-maintenance, belief revision, belief update and planning. In this paper, we add default reasoning to this list, by showing how to represent default logic theories in CHR$^\vee$. We then discuss how to leverage this representation together with the well-know correspondence between default logic and *Negation As Failure* (NAF) in logic programming, to propose an extension CHR$^{\vee,naf}$ of CHR$^\vee$ allowing NAF in the rule heads.

## 1 Introduction

CHR$^\vee$ [1] is a first-order, relational rule language that incorporates forward chaining of conditional rewrite rules and guarded production rules, with logic programming's backtracking search of disjunctive alternatives in the rules' right-hand side (called "body" in CHR$^\vee$).

It was initially conceived to declaratively implement constraint solving tasks by harmoniously integrating three techniques widely used for this task: (a) constraint simplification using conditional rewrite rules (called "simplification rules" in CHR$^\vee$), (b) constraint propagation using guarded production rules (called "propagation rules" in CHR$^\vee$), and (c) finite domain constraint search using disjunctive rules (either simplification or propagation rules).

This integration turned out to be powerful enough to use CHR$^\vee$ for a surprisingly wide variety of other reasoning tasks beyond constraint solving: classical deduction [1], description logics, concept subsumption and individual classification [2], abduction [3], truth-maintenance [4], belief revision [5], belief update and planning [6].

In this paper, we first show how to add default reasoning to the list of automated reasoning tasks that can be performed by reusing a CHR$^\vee$ inference engine. Our proposal is based on a mapping of a Default Logic Theory[7] into a CHR$^\vee$ rule base. We then leverage this mapping, together with the well-known correspondence between default reasoning and logic programming with Negation As Failure (NAF) [8], to propose an extension CHR$^{\vee,naf}$ of CHR$^\vee$ that allows the `naf` connective in rule heads. We also explain how our CHR$^{\vee,naf}$ proposal semantically differs from the CHR$^\neg$ Negation As Absence (NAA) proposal [9] and why we believe its applicability is wider.

The contributions of this paper are (a) mapping of default logic formulas into CHR$^\vee$ bases, and (b) leveraging it to extend CHR$^\vee$ with NAF. This is very significant due to (a) the pervasive utility of default reasoning and NAF in artificial intelligence applications, and (b) their well-studied relations with other forms of non-monotonic reasoning abduction [10], truth-maintenance [10], belief revision [11] and inheritance with overriding [12]. Each of these reasoning tasks must be carried out by any agent that acts in a partially observable environment, the most common case in practical applications.

The rest of this paper is organized as follows. In the next sections, we briefly review in turn the syntax and semantics of Default Logic and CHR$^\vee$. In section four we present our mapping of the former to the latter. In section five, we show how to leverage such mapping to extend CHR$^\vee$ with NAF. In section six, we discuss related work in non-monotonic reasoning in CHR$^\vee$, before concluding in section seven.

## 2 Default Logic

Default Logic [13][14][7] formalizes the reasoning of an agent in a partially observable environment, where it misses some volatile knowledge, typically the truth value of a fluent[3], that is essential to choose its next action. In such situation, the agent needs to base its choice on some default hypothesis about the truth value of that fluent. While this hypothesis must be consistent with the agent's current volatile knowledge about the current state of the environment, it can nevertheless be deductively unsound and thus subject to revision upon subsequent deduction from reliable new sensor information of contradictory volatile knowledge.

Such reasoning cannot be appropriately formulated directly in Classical First-Order Logic (CFOL) due to the knowledge monotonicity assumption of this formalism. This is illustrated by the following example:

*Example 1.* Let us assume that we want to represent the following piece of knowledge: *Birds typically fly, Penguins and Albatrosses are birds, Penguins do not fly and Tux is a Penguin.*

The CFOL formula $\forall x((Bird(x) \rightarrow Flies(x)) \wedge (Penguin(x) \rightarrow Bird(x)) \wedge Penguin(tux) \wedge (Penguin(x) \rightarrow \neg Flies(x)) \wedge (Albatross(x) \rightarrow Bird(x)) \wedge \neg(Penguin(x) \wedge Albatross(x)))$ does not properly represent such knowledge because it entails $\bot$ due to the inability of pure deduction in CFOL to retract the conclusion $Flies(tux)$ entailed by the first three clauses of the formula in the light of the more specific conclusion $\neg Flies(tux)$ entailed by the third and fourth clauses. The *core* problem of CFOL is the inability to represent rules with exceptions with the only two quantifiers of CFOL: universal and existential. Default logic extends CFOL with default inference rules that capture such rules with exception. It is formally defined below.

[3] A property of the environment that changes over time or due to the actions executed by the agent.

**Definition 1 (Default Logic Theory).** *A Default Logic Theory is a tuple* $\langle D, W\rangle$ *where* $D$ *is the set of default rules and* $W$ *is a set of CFOL formulas. Each default rule assumes the following form:*

$$\frac{\alpha : \beta}{\gamma}$$

where $\alpha$ (*prerequisite*), $\beta$ (*justification*) and $\gamma$ (*conclusion*) are CFOL formulas. The intended meaning of this default rule is:

*If* $\alpha$ *is entailed by the current knowledge base* ($KB \models \alpha$) *and* $\beta$ *is consistent with the current knowledge base* ($KB \wedge \beta \not\models \bot$) *then* $\gamma$ *can be assumed.*

The *extension* $\varepsilon$ of a Default Theory is the maximal set of formulas that can be derived and assumed by default from it. This concept is formally defined in the Definition 3. Notice that a Default Theory may have one, many or no extension at all.

**Definition 2 (Deductive Closure).** *Let* $T$ *be a set of CFOL formulas. The* Deductive Closure *of* $T$ *is a set* $Th(T)$ *such that* $T \subseteq Th(T)$ *and for each* $p \in Th(T)$, *if* $p \models q$ *then* $q \in Th(T)$.

**Definition 3 (Extension of a Default Theory).** *We define* $\varepsilon$ *as an extension for the default theory* $\langle D, W\rangle$ *if and only if it satisfies the following equations:*

$$E_0 = W$$

*and for* $i > 0$,

$$E_{i+1} = Th(E_i) \cup \left\{\gamma | \frac{\alpha : \beta}{\gamma} \in D, \alpha \in E_i, \neg\beta \notin \varepsilon\right\}$$

*and,*

$$\varepsilon = \bigcup_{i=0}^{\infty} E_i$$

*Example 2.* Let us show how to model the knowledge in the Example 1 as a Default Theory. At first, the default rules:

$$D = \left\{ \frac{Bird(x) : Flies(x)}{Flies(x)}, \frac{Bird(x) : Penguin(x)}{Penguin(x)}, \frac{Bird(x) : Albatross(x)}{Albatross(x)} \right\}$$

Notice that, when we state this knowledge as default rules, we emphasize what is assumed by hypothesis. We are now going to represent the logical formulas of our theory:

$$W = \{Penguin(tux), Penguin(x) \rightarrow \neg Flies(x), \neg(Penguin(x) \wedge Albatross(x))\}$$

Two of the possible extensions for this theory are:

$$\varepsilon_1 = W \cup \{Flies(x), Albatross(x)\}$$

and,

$$\varepsilon_2 = W \cup \{\neg Flies(x), Penguin(x)\}$$

### 2.1 NAF as Default Logics

In this section we show how to express NAF by means of Default Theories. This negation is different from the usual CFOL one. In order to avoid misinterpretations we utilize the symbol $naf$ for negation as failure and *cneg* for classical logical negation.

Take the following rules:

$$p \leftarrow cneg(q).$$

$$p' \leftarrow naf(q).$$

In the first case, $p$ can be proved only if $q$ can be *proved* to be false. In the second one, $p'$ can be assumed to be true if $q$ cannot be proved to be true. The difference relies on the fact that $q$ may be true, but unknown (i.e., it may not be possible to deduce it). In the first example it is not possible to deduce $p$ (because it is not possible prove $q$ false), but in the second example it is possible to deduce $p'$ (because it is not possible to prove $q$ true).

In [13], Grigoris Antoniou shows how to model NAF as Default Theories in a natural way. The idea is to add a default rule like the following one for each ground fact $\phi$:

$$\frac{: naf\phi}{naf\phi}$$

This means that if the hypothesis $naf\phi$ is consistent (in other words, if $\phi$ can't be proved), $naf\phi$ can be assumed.

## 3 CHR$^\vee$

*Constraint Handling Rules with Disjunction* (CHR$^\vee$) [15] is a first-order, relational rule language for writing Constraint Solvers.

There are two kinds of constraints: the *user defined* and the *built-ins*. The first set is formed by the constraints whose semantics is given by the set of rules

and the second set is formed by the constraints whose semantics is provided by the inference engine.

There are three kinds of rules in CHR$^\vee$: simplification, propagation and simpagation. They can be respectively described as follows:

$$r@H_r \Leftrightarrow G|B.$$

$$s@H_k \Rightarrow G|B.$$

$$t@H_k \backslash H_r \Leftrightarrow G|B.$$

In this example, `r`, `s` and `t` are identifiers for the rules and can be omitted. $H_r$ and $H_k$ are the heads of the rules. More specifically, $H_k$ are the *kept heads*, which are kept in the constraint store; and $H_r$ are the *removed heads*, which are removed from it. $G$ is the guard and $B$ is the body. If the guard is `true`, it can be omitted.

The abstract operational semantics for CHR$^\vee$ is defined as a transition system. A CHR$^\vee$ state is a disjunction of one of more *subgoals* which are conjunctions of user defined constraints, built-ins or disjunctions. A state is called final if no transition is applicable or all of its subgoals are inconsistent (in this case, it is called *failed*). For more details see [15].

The following diagram presents the transition rules of CHR$^\vee$:

**Solve**
If $CT \models \forall(S \Leftrightarrow S')$ and $S'$ is the normal form of $S$
then $S \mapsto_P S'$

**Propagate**
If $(H \Rightarrow G|B)$ is a fresh variant of a rule with variables $\bar{x}$
and $CT \models \forall(S \rightarrow \exists\bar{x}(H = H' \wedge G)$
then $(H' \wedge S) \mapsto_P (H = H' \wedge B \wedge G \wedge H' \wedge S)$

**Simplify**
If $(H \Leftrightarrow G|B)$ is a fresh variant of a rule with variables $\bar{x}$
and $CT \models \forall(S \rightarrow \exists\bar{x}(H = H' \wedge G)$
then $(H' \wedge S) \mapsto_P (H = H' \wedge B \wedge G \wedge S)$

**Simpagate**
If $(H_k \backslash H_R \Leftrightarrow G|B)$ is a fresh variant of a rule with variables $\bar{x}$
and $CT \models \forall(S \rightarrow \exists\bar{x}(H_k = H'_k \wedge H_R = H'_R \wedge G)$
then $(H'_k \wedge H'_R \wedge S) \mapsto_P (H_k = H'_k \wedge H_R = H'_R \wedge B \wedge G \wedge H'_k \wedge S)$

**Split**
$(S_1 \vee S_2) \wedge S \mapsto_P (S_1 \wedge S) \vee (S_2 \wedge S)$

## 4 Describing Default Logic Theories in CHR$^\vee$

In this Section we describe our approach to describing Default Theories as CHR$^\vee$ rule bases. It consists of mapping each default rule into two sets of propagation rules. Let us take the following default rule:

$$\frac{\alpha : \beta}{\gamma}$$

Without loss of generality, consider $\alpha$ and $\gamma$ as being in the Conjunctive Normal Form (CNF) and $\beta$ as being in the Disjunctive Normal Form (DNF) [4]:

$$\alpha = (\alpha_{1,1} \wedge \ldots \wedge \alpha_{n,1}) \vee \ldots \vee (\alpha_{1,m} \wedge \ldots \wedge \alpha_{n,m})$$

$$\beta = (\beta_{1,1} \vee \ldots \vee \beta_{n,1}) \wedge \ldots \wedge (\beta_{1,m} \vee \ldots \vee \beta_{n,m})$$

$$\gamma = (\gamma_{1,1} \wedge \ldots \wedge \gamma_{n,1}) \vee \ldots \vee (\gamma_{1,m} \wedge \ldots \wedge \gamma_{n,m})$$

**Definition 4 (Search Rules).** *Given a default rule $r$ as above, we define search($r$) as the set containing the following rules:*

r1 @ $\alpha_{11}$, ..., $\alpha_{n1}$ ==> (r, $\gamma$) ; true.
...
rn @ $\alpha_{1m}$, ..., $\alpha_{nm}$ ==> (r, $\gamma$) ; true.

The intended meaning of these rules is: if $\alpha$, or any of its disjunctive components, is in the Constraint Store (and is therefore entailed by it), we have two options:

- assume $\beta$ and add $\gamma$ to the Constraint Store,
- do not assume $\beta$.

The new constraint `r` has exactly the meaning of *$\beta$ is assumed by default*. This can be accomplished by adding $\beta$ to the constraint store. However, when dealing with simplification and simpagation rules, part of $\beta$ may be removed by some simplification or simpagation rule and it might not be possible to prove its falsehood in the future.

**Definition 5 (Integrity Rules).** *Given a default rule $r$ as above, we define integrity($r$) as the set containing the following rules:*

s1 @ r, cneg_$\beta_{11}$, ..., cneg_$\beta_{n1}$ ==> false.
...
sn @ r, cneg_$\beta_{1m}$, ..., cneg_$\beta_{nm}$ ==> false.

[4] Notice that we employ Abdennadher's notation for negation as defined in [3], in which a negated constraint $cneg(\phi)$ is represented by a new constraint $cneg\_\phi$ and a integrity constraint $\phi, cneg\_\phi \Rightarrow false$.

The intended meaning of this set of rules is: if *cneg_*$\beta$ can be proved, or any of its components, and $\beta$ has been assumed, then the store is inconsistent.

The idea behind this solution is to use CHR$^\vee$ as a platform for *searching* for an extension for the Default Theory. For example, let us return to the Example 1. The result of the application of the transformation to this problem is:

```
r1 @ bird(X) ==> (r1(X), flies(X)) ; true.
r2 @ bird(X) ==> (r2(X), penguin(X)) ; true.
r3 @ bird(X) ==> (r3(X), albatross(X)) ; true.

s1 @ r1(X), cneg_flies(X)     ==> false.
s2 @ r2(X), cneg_penguin(X)   ==> false.
s3 @ r3(X), cneg_albatross(X) ==> false.
```

Let us feed the CHR$^\vee$ engine with the following set of rules and initial constraint store:

```
penguin(X) ==> cneg_flies(X).
penguin(X), albatross(X) ==> false(X).

query: bird(tux).
```

The following set of final states is going to be obtained (omitting the new constraints added by the transformation):

```
S1 = {  bird(tux), albatross(tux), flies(tux)  }

S2 = {  bird(tux), albatross(tux)  }

S3 = {  bird(tux), penguin(tux), cneg_flies(tux)  }

S4 = {  bird(tux), flies(tux)  }

S5 = {  bird(tux) }
```

Notice that each store corresponds to a set of assumed hypotheses, ranging from two hypotheses in `S1` and `S3` to no hypothesis in `S5`.

We are now going to demonstrate that our approach successfully computes all correct extensions, at least for a restricted set of Default Theories. We call this restricted set of *Propositional CHR Propagation Restricted Default Theories*.

**Definition 6 (Propositional CHR Default Theory).** *A default theory* $\langle D, W\rangle$ *is said to be a* Propositional CHR Default Theory*, if*

- *D is composed of default rules of the form, where* $\alpha$ *and* $\gamma$ *are in the CNF and* $\beta$ *is in the DNF, and are propositional CFOL formulas:*

$$\frac{\alpha : \beta}{\gamma}$$

- *$W$ is composed of a conjunction of:*
  - *A set of atomic constraints,*
  - *A set of logical rules of the form $H \wedge G \rightarrow B$ or $G \rightarrow (H \leftrightarrow B)$, equivalent to CHR propagation and simplification rules without disjunctions, respectively.*
  - *For each constraint $\phi$ there is a rule of the form: $\phi \wedge cneg\_\phi \rightarrow \bot$*

**Definition 7 (Propositional CHR Propagation Restricted Default Theories).** *A default theory $\langle D, W\rangle$ is said to be* Propositional CHR Propagation Restricted *, if and only if it is a Propositional CHR Default Theory and the rules in $W$ are equivalent to non-disjunctive CHR propagation rules.*

Our notion of *equivalence* of CHR states and Default Theories extensions is outlined by the Definition 8. It captures the fact that the new constraints introduced by our approach do not change the meaning of a state.

**Definition 8 (Equivalence of States and Theories).** *Let $\langle W, D\rangle$ be a Propositional CHR Propagation Restricted Default Theory, and let $W$ be the conjunction of the constraints in a CHR$^\vee$ state $S$ and $P$ a set of CHR rules. We say that $S$ is equivalent to $W$ if $W$ contains the logical meanings of the rules in $P$, the constraints in $S$ and no other instance of a constraint appearing in $P$ or in $S$.*

**Theorem 1.** *Let $\langle D, W\rangle$ be a Propositional CHR Propagation Restricted Default Theory. Let $W$ be the conjunction of the initial goal $S$ and the set of CHR rules $P$, and $R$ the set obtained by transforming the default rules in $D$ into CHR rules. For every extension $\varepsilon$ of $\langle D, W\rangle$, if $E_i$ is equivalent to a subgoal $S'_g$ of some state $S'$ (such that $S \mapsto_{P\cup R} \ldots \mapsto_{P\cup R} S'$ is a finite derivation), then there exists an state $S''$ such that $E_{i+1}$ is equivalent to a subgoal $S''_g$ of $S''$ and $S' \mapsto_{P\cup R} \ldots \mapsto_{P\cup R} S''$ is a finite derivation for it.*

*Proof (Sketch).* If $\varepsilon$ is an extension and the default rule $r = \frac{\alpha:\beta}{\gamma}$ is applied between the step $E_i$ and $E_{i+1}$. Since the extension exists, $\beta$ is consistent with it. It's easy to see that we can divide the derivation between $S'$ and $S''$ into two steps: (i) compute the deductive closure of $S'_g$ and (ii) execute some of the rules in $search(r)$. Applying all the derivation steps to the subgoal $S'_g$ will lead us to a state $S''$ containing a subgoal $S''_g$ which is equivalent to $E_{i+1}$.

□

**Theorem 2 (Completeness).** *Let $\langle D, W\rangle$ be a Propositional CHR Propagation Restricted Default Theory. Let $W$ be the conjunction of the initial goal $S$ and the set of CHR rules $P$, and $R$ the set obtained by transforming the default rules in $D$ into CHR rules. For each non-failed derivation $S \mapsto^*_{P\cup R} S_f$, all extensions $\varepsilon$ of $\langle D, W\rangle$ are subgoals of $S_f$.*

*Proof.* By contradiction. Let us suppose there exists an extension $\varepsilon$ which is not a subgoal of $S_f$. By definition, $\varepsilon = \bigcup_{i=0}^{\infty} E_i$. By Theorem 1, it follows easily that every $E_i$ should be equivalent to a subgoal of $S_f$, and thus $\varepsilon$ should also be a subgoal of $S_f$.

□

**Theorem 3 (Weak Correctness).** *Let $\langle D, W\rangle$ be a Propositional CHR Propagation Restricted Default Theory. Let $W$ be the conjunction of the initial goal $S$ and the set of CHR rules $P$, and $R$ the set obtained by transforming the default rules in $D$ into CHR rules. For each non-failed derivation $S \mapsto^*_{P\cup R} S_f$, every non-failed subgoal of $S_f$ is equivalent to a subset of an extension $\varepsilon$ of $\langle D, W\rangle$.*

*Proof (Sketch).* By induction on the derivation length. The initial constraint store is equivalent to a subset of every extension, by definition. By the proof of the Theorem 1 it is easy to verify that each transition in $S \mapsto^*_{P\cup R} S_f$ is one step of the computation of a deductive closure or in the execution of a default rule between some step $E_i$ and $E_{i+1}$ in the derivation of an extension $\varepsilon$.

□

If we try to extend these results to less restricted versions of the Theorems 2 and 3 we are going to see that both properties are going to be lost, mainly due to the fact that the simpagation and simplification actually remove part of the state. Therefore, the obtained subgoals are not going to be complete extensions anymore.

Another possible extension is allowing disjunctive bodies. In this case, an extension is not going to correspond to a subgoal, but to a set of subgoals, which can be easily computed, each subgoal contains an extra constraints for the assumed hypothesis: each explanation is the disjunction of the subgoals relying on the same hypotheses.

## 5 CHR$^{\vee,naf}$: Extending CHR$^{\vee}$ with NAF in the rule heads

In this Section, we are going to present CHR$^{\vee,naf}$, an extension for CHR with negated rule heads. In this extension, there are three kinds of rules: simpagation, propagation and simplification. The simpagation rules generalize all of them. Their general syntax is:

$$r@H_k \backslash H_r \backslash\backslash N_1|G_1 \backslash\backslash \ldots \backslash\backslash N_n|G_n \Leftrightarrow G|B.$$

In this example, $r$ is an identifier for the rule, which can be omitted. $H_k$ are the *kept heads*, which are kept in the constraint store when the rule fires. The $H_r$ are the *removed heads*, which are removed when the rule fires. $H_k$ and $H_r$ are the *positive heads* of the rule, whereas $N_1, \ldots, N_n$ are the *negative heads*. All heads must contain only user defined constraints. $G_1, \ldots, G_n$ are the negated guards, and like the positive guard $G$, may be empty and in this case can be omitted.

Any variable introduced in a guard cannot be used in another guard, i.e., the guards can only use the variables appearing in the positive heads and in their corresponding negative head. The variables defined by the negative guards cannot appear in the rule body and all guards are composed only of built-in constraints.

If $H_k$ is empty, it can be omitted (along with the following *backslash*) and the rule is called a *simplification* rule. If $H_r$ is empty, it can also be omitted (along with the preceding *backslash*) and the sign $\Leftrightarrow$ is changed to $\Rightarrow$. This rule is called *propagation* rule. The negative heads cannot be empty. A rule is authorized to have from 0 to any number of negated heads.

Finally, $B$ is the rule body, and is composed of a disjunction of conjunctions of user defined and built-in constraints. No variable defined in a negative head can appear in the rule body.

For now, let us consider the semantics of a rule such that as being:

$$\forall G \rightarrow (H_k \wedge H_r \wedge naf(\exists((N_1 \wedge G_1) \wedge \ldots \wedge (N_n \wedge G_n))) \leftrightarrow B)$$

In other words: *if the head is in the constraint store and there is no proof that the negated head is inconsistent with it, we can assume it by hypothesis.*

*Example 3.* Let us suppose we want to find the minimum value `X` for which there exists a `c(X)` in the constraint store.

The common approach is to do something like:

```
c(X) \ getMin(Min) <=> current(X,Min).
c(X) \ current(Current,Min) <=> X<Current | current(X,Min).
current(Current, Min) <=> Min = Current.
```

Notice that this solution explores the refined operational semantics of CHR and it is inherently *not* confluent. In a isolated piece of code like this these properties might not cause grave problems. However, in large rule bases, confluence problems may be much harder to solve. We want, as much as possible, to find confluent solutions to the problems.

In CHR$^{\vee,naf}$, this example is implemented much simply, by the means of following rule:

```
getMin(M), c(X) \\ c(Y) | Y < X ==> M = X.
```

The logical reading of this rule is:

$$\forall X, M(getMin(M) \wedge c(X) \wedge naf(\exists Y(c(Y) \wedge Y < X)) \rightarrow X = M)$$

### 5.1 Negation in Rule Heads as Default Reasoning

Now we show how to extract a Default Theory from a CHR$^{\vee,naf}$ rule base. Let us take the CHR$^{\vee,naf}$ rule from the Example 3:

```
getMin(M), c(X) \\ c(Y) | Y < X ==> X = M.
```

It is possible to translate the logical reading of this rule in the following default rule:

$$\forall X, Y, M \frac{getMin(M), c(X) : \neg(c(Y) \wedge Y < X)}{X = M}$$

From this example, it is possible to infer that every CHR$^{\vee,naf}$ propagation rule can be naturally translated into a default rule. The general pattern is that the following rule:

$$r@H \backslash\backslash N_1 | G_1 \backslash\backslash \ldots \backslash\backslash N_n | G_n \Rightarrow G | B.$$

generates the following default rule:

$$\forall \frac{H \wedge G : (\neg(N_1 \wedge G_1) \wedge \ldots \wedge \neg(N_n \wedge G_n))}{B}$$

Unfortunately, this translation does not account either for general simpagation rules or for simplification rules. What is missing is the capability of removing the constraints in the head from the constraint store.

This can be easily accomplished by mapping each simpagation or simplification rule into a pair of rules, one for propagating the body and another for removing the head.

For example, let us take the following simpagation rule:

```
r @ a(X) \ b(Y) <=> g(Z) | c.
```

It is possible to rewrite it into an equivalent pair of rules:

```
r1 @ a(X), b(Y) ==> g(Z) | s(X,Y,Z).
r2 @ s(X,Y,Z), b(Y) <=> c.
```

This pair of rules is equivalent to former one (in the sense of the Definition 8). Since the negated heads in a CHR$^{\vee,naf}$ rule base act like a precondition, this strategy can be extended to CHR$^{\vee,naf}$ programs. The idea is than translate each simpagation rule of the form:

$$r@H_k \backslash H_r \backslash\backslash N_1 | G_1 \backslash\backslash \ldots \backslash\backslash N_n | G_n \Leftrightarrow G | B.$$

to a pair of rules, a propagation and a simplification rule:

$$r_1@H_k, H_r \backslash\backslash N_1 | G_1 \backslash\backslash \ldots \backslash\backslash N_n | G_n \Rightarrow G | s.$$

$$r_2@s, H_r \Leftrightarrow B.$$

The next step is then, transforming the rule $r_1$ into a default rule, the result is:

$$\forall \frac{H_k \wedge H_r \wedge G : \neg(N_1 \wedge G_1) \wedge \ldots \wedge \neg(N_n \wedge G_n)}{s}$$

The following set of CHR$^\vee$ rules is obtained[5].

```
r21 @ Hk, Hr ==> G | (r, s) ; true.
r22 @ r, N1 ==> G1 | false.
...
r2n @ r, Nn ==> Gn | false.
```

Let us return to the Example 3. The complete transformed rule base is as follows:

```
 r1 @ getMin(M), c(X) ==> (r(X,M), X = M) ; true.
 r2 @ r(X,M), c(Y) ==> Y < X | false.
```

Let us suppose we feed the CHR$^\vee$ inference engine with the following goal:

```
c(3), c(9), getMin(M)
```

The two final Constraint Stores are:

```
S1 = { c(3), r(3,3), c(9), getMin(3) }
S2 = { c(3) , c(9), getMin(M) }
```

The first one is obtained by assuming `c(3)` as the minimum and the other one is obtained by assuming no constraint as the minimum.

## 6 Related Work

### 6.1 Abduction in CHR$^\vee$

In [10], Kakas et al show that Default Logics is a special case of Abduction. They say that the process of assuming hypotheses can be viewed as a form of abduction, where instances of defaults are the candidate abducibles.

In [3], Abdennadher explains how to utilize CHR$^\vee$ as a platform for Abductive Reasoning and presents a method for expressing abductive problems as CHR$^\vee$ rule bases.

In theory, it is possible to combine both approaches in order to reason about Default Theories in CHR$^\vee$. The main advantage of our approach is that it allows the addition of default rules to existing CHR$^\vee$ rule bases, while the hybrid approach combining [10] and [3] would require the translation of the existing rule bases into abductive problems and then the translation of these problems into CHR$^\vee$, which might not be trivial.

[5] Notice that we map former guards into guards in the new rules.

### 6.2 Comparing CHR$^{\neg}$ and CHR$^{\vee,naf}$

Both CHR$^{\neg}$ and CHR$^{\vee,naf}$ share the same syntax, but differ substantially in their semantics. The semantics for CHR$^{\neg}$ was based on the refined operational semantics for CHR defined in [16] and consisted of restricting the applicability of CHR rules to situations where no negated head were present and adding the notion of *Triggering on removal*, in which, a rule should also fire when a negated constraint is removed from the constraint store.

That semantics presented some undesirable features, which this one aims to overcome. The first of these effects is the lost logical reading, in the sense that, because of its essentially operational semantics, it is not anymore possible to map each rule into a logical formula. The present semantics brings back a logical reading to rules with negative heads, in the sense that each rule can be read as a *default rule*, where the pre-requisites are in the positive head, the justification is the negative head and the conclusion is the body.

Another undesired feature is the unexpected behavior for some programs. The operational semantics for CHR$^{\neg}$ turned out to lead to counter-intuitive results for some simple programs, as the one described in the Example 4.

*Example 4 (Order).* Under CHR$^{\neg}$, negatively occurring constraints have to be added in the right order. In the following rule base, everytime the first rule fires, the child is declared an orphan. The reason for that is the fact that when the constraint `child(C)` is added to the constraint store and the constraints `father(F,C)` and `mother(M,C)` have not yet been added, and thus, the second rule fires.

```
birth(C,F,M) <=> child(C), father(F,C), mother(M,C).
child(C) \\ father(_,C) \\ mother(_,C) ==> orphan(C).
```

To illustrate the difference between both semantics, let us suppose we try the following initial constraint store:

```
birth(a, b,c), child(e).
```

In this example, `a` is not orphan, but we don't know whether `e` is. We are going to obtain two final constraint stores:

```
S1 = { child(a), father(b, a), mother(c, a), child(e), orphan(e) }
S2 = { child(a), father(b, a), mother(c, a), child(e) }
```

The first one, assumes `e` to be an orphan, and the second one does not assume anything. In CHR$^{\neg}$, the final constraint store for this initial constraint store is:

```
S1 = { child(a), father(b, a),
       mother(c, a), child(e),
       orphan(a), orphan(e) }
```

This result is unexpected because `a` is clearly not orphan.

## 7 Conclusion

At this work, we confirmed the flexibility of the CHR$^\vee$ language by presenting it as a platform for Default Reasoning services. We defined an approach that permits us to rewrite Default Rules as CHR$^\vee$ propagation rules and reuse the built-in search capabilities of CHR$^\vee$ in order to find consistent sets of hypotheses that can be assumed in a given Default Theory.

We have also investigated how to leverage the correspondence between Default Logic and *Negation As Failure* (NAF) in order to propose an extension CHR$^{\vee,naf}$ for CHR$^\vee$ allowing negated constraints and guards in the rule head. We showed how this extension relate to CHR$^\neg$ [9], which employs an operational concept of negation: *Negation As Absence* (NAA).

We propose the following future works:

- **Triggering on Removal:** This is an important feature of CHR$^\neg$ which is not supported by CHR$^{\vee,naf}$ because it is not declarative. In order to allow this kind of reasoning we would need to employ some better-founded semantics for removal, like the one employed by Adaptive CHR$^\vee$ [4].

- **Complexity of Default Logics in CHR$^\vee$:** As pointed out by [14], the problem of enumerating all the extensions for a Default Theory has an exponential time complexity. This is easily shown by the fact that each possible hypothesis generates two possibilities: considering it and not considering it. Under this context, it is easy to notice that the number of states computed by the CHR$^\vee$ machine increases exponentially with the input size.

  In fact, only some of the returned states are complete extensions. For example, in the list of final states presented at the example in the Section 4, only the first one was really an extension. One of the future works is to develop an operational semantics taking a bias in the hypothesis into accounts, making it possible to prioritize the returned solutions.

  This new strategy will not be able to reduce its worst case complexity, but will improve its average time complexity.

- **Stronger Theoretical Results:** the proofs presented at the Section 2.1 cover only a very limited range of Default Theories (the Propositional CHR Propagation Restricted Default Theories). A future work is to extend this results, initially to Default Theories with variables and quantifiers and then to simpagation and simplification rules.

## References

1. Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Springer-Verlag (2003)

2. Frühwirth, T.: Description Logic and Rules the CHR Way. Fourth Workshop on Constraint Handling Rules (2007) 49–62
3. Abdennadher, S.: Rule-Based Constraint Programming: Theory and Practice. Technical report, Institut für Informatik, Ludwig-Maximilians-Universität München (July 2001)
4. Wolf, A., Robin, J., Vitorino, J.: Adaptive CHR meets CHR$^\vee$: An Extended Refined Operational Semantics for CHR$^\vee$ based on Justifications. In: Proceedings of the Fourth Workshop on Constraint Handling Rules (CHR 2007), Porto, Portugal (2007) 1–15
5. Jin, Y., Thielscher, M.: Representing beliefs in the fluent calculus. In: ECAI. (2004) 823–827
6. Thielscher, M.: Reasoning Robots: The Art and Science of Programming Reasoning Agents. Applied Logic Series 5. Kluwer (2005)
7. Reiter, R.: Readings in nonmonotonic reasoning. Morgan Kaufmann Publishers Inc. (1980)
8. Fages, F.: Consistency of Clark's Completion and Existence of Stable Models. Methods of Logic in Computer Science (1994) 1:51–60
9. Weert, P.V., Sneyers, J., Schrijvers, T., Demoen, B.: Extending CHR with Negation as Absence. Third Workshop on Constraint Handling Rules (2006) 125–140
10. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive logic programming. Journal of Logic and Computation **2**(6) (1992) 719–770
11. Pagnucco, M.: The Role of Abductive Reasoning within the Process of Belief revision. Technical report, University of Sydney (1996)
12. Yang, G., Kifer, M.: Inheritance in Rule-Based Frame Systems: Semantics and Inference. Journal on Data Semantics (JoDS) **II** (2006) 79–135
13. Antoniou, G.: A tutorial on default logics. ACM Computing Surveys **31**(4) (1999) 337–359
14. Eiter, T., Gottlob, G.: Semantics and complexity of abduction from default theories. In Mellish, C., ed.: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, San Francisco, Morgan Kaufmann (1995) 870–877
15. Abdennadher, S.: A Language for Experimenting with Declarative Paradigms. Second Workshop on Rule-Based Constraint Reasoning and Programming (2000)
16. Duck, G.J., Stuckey, P., de la Banda, M.G., Holzbaur, C.: The Refined Operational Semantics of Constraint Handling Rules. In: Proceedings of the 20th International Conference on Logic Programing (ICLP'04), Saint-Malo, France, Springer Berlin / Heidelberg (2004) 90–104

# Relating Constraint Handling Rules to Datalog

Beata Sarna-Starosta[1], David Zook[2], Emir Pasalic[2], and Molham Aref[2]

[1] Department of Computer Science & Engineering
Michigan State University, East Lansing, MI 48824,
Logic Blox, Atlanta, GA 30309
E-mail: bss@cse.msu.edu
[2] Logic Blox, Atlanta, GA 30309
E-mail: {david.zook,emir.pasalic,molham.aref}@logicblox.com

**Abstract.** Datalog$^{\mathrm{LB}}$ is an extension of Datalog supporting global stratification of negation and functional dependencies, designed for use in industrial-scale decision automation applications. Constraint Handling Rules (CHR) is a declarative rule-based programming language, particularly suitable for specifying custom constraint solvers at a high level. Our goal is to enhance Datalog$^{\mathrm{LB}}$ with CHR-like capabilities in order to improve its expressive power and open it to specification of general-purpose constraint solvers for industrial applications. In this paper we relate the two formalisms and define a translation of a significant class of CHR programs into Datalog$^{\mathrm{LB}}$. It turns out that the translation enables reasoning about the properties of CHR programs at a high level of Datalog logic.

## 1 Introduction

Constraint Handling Rules (CHR) [1] is a declarative formalism of multi-headed, committed-choice, guarded, multiset rewriting rules, originally designed for extending host languages, such as Prolog or Haskell, with user-defined constraint solvers. The expressiveness of CHR facilitates specification of a wide variety of problems at a high level, and its clean semantics supports program analysis and transformation, enabling non-trivial performance improvements. Thus, the formalism has evolved into a general-purpose programming language, with application domains including computational linguistics, software engineering, deductive databases, semantic web, and more.

We consider CHR in the context of its potential benefits for the domain of automated decision support. We are developing a framework for building configurable decision support systems, based on a reasoning language rooted in Datalog. Although more powerful than pure Datalog, our language—called Datalog$^{\mathrm{LB}}$—often lacks the flexibility and generality to define constraint solvers for industrial applications. Thus, we aim to port some of the features of CHR to Datalog$^{\mathrm{LB}}$ in order to improve the latter's expressive power, and to make our systems capable of handling more complex problems.

This paper reports our experience from the initial step towards porting the functionality of CHR to Datalog$^{\mathrm{LB}}$, in which we establish a relationship between the two formalisms. The benefit of this step is mutual. On one hand, by exposing aspects of CHR that exceed Datalog$^{\mathrm{LB}}$'s handling capabilities, it fosters better design decisions leading to the improvement of our language. On the other hand, by demarcating the classes of

CHR programs that map into pure Datalog and Datalog$^{\mathrm{LB}}$, it allows reasoning about the properties of these program classes at the level of the declarative and well-studied Datalog logic. In summary, our paper makes the following contributions:

- it formally defines a basic translation schema from CHR to pure Datalog
- it identifies a class of CHR programs that are amenable for the basic translation, and thus share properties with pure-Datalog programs
- it extends the basic translation schema to accommodate properties of CHR not compliant with pure Datalog, but expressible in Datalog$^{\mathrm{LB}}$
- it identifies a class of CHR programs that are amenable for the extended translation, and thus share properties with Datalog$^{\mathrm{LB}}$ programs.

We illustrate different stages of our translation with examples of CHR programs and their pure-Datalog or Datalog$^{\mathrm{LB}}$ counterparts. All CHR programs are either borrowed or adapted from the WebCHR online demo [2].

In the remainder of the paper, Section 2 provides background on Datalog, Datalog$^{\mathrm{LB}}$ and CHR; Section 3 defines the basic translation schema from a restricted subclass of CHR to pure Datalog; Section 4 extends the basic translation schema onto a larger class of CHR and Datalog$^{\mathrm{LB}}$; Section 5 considers some of the challenges of the unrestricted CHR translation; and Section 6 concludes with a discussion and review of related work.

## 2 Preliminaries

We use standard notions of variables, constants, $n$-ary functions, terms and clauses [3], and of databases and the relational algebra [4]. We use, possibly subscripted, upper-case letters to denote sets and sequences of entities, and lower-case letters to denote their elements. Usually, $t$ refers to a term in general, $c$ refers to a constraint term, with a constraint symbol at the root, and $d$ refers to a Datalog clause. We use $\mathit{vars}(t)$ to denote the set of variables in a term $t$. We extend the functions defined over elements of the collections onto the entire collections whenever needed and obvious from the context.

### 2.1 Datalog

Datalog is a subset of Prolog developed in 1970s for deductive databases. The original specification of Datalog, which we call *pure Datalog*, extended traditional database query languages with support for recursion, at the same time avoiding Prolog's non-termination issues, and thus conforming to the set-at-a-time reasoning scheme of the relational algebra. Syntactically, pure Datalog coincides with Prolog restricted so that: (i) unit clauses (facts) are always ground, (ii) all predicate arguments are variables or constants, and (iii) it is negation-free. To ensure that the condition (i) is satisfied, all clauses must be *safe*, i.e., every variable in a clause must appear in the clause's body.

The specification of pure Datalog is too confining for many practical applications, and numerous extensions have been proposed to relax its restrictions. Most notably, these include different models for handling negation, admitting disjunction in clause heads, and support for object-oriented programming. In recent years, Datalog received

attention of researchers from areas such as program analysis [5], networks [6], security protocols [7], knowledge representation [8], robotics [9], games [10], and more.

The power of Datalog lies in its ability to define new predicates, which are calculated in terms of given data. A Datalog program operates on two disjoint sets of predicates: the *extensional database (EDB)*—the predicates defined by externally supplied facts—and the *intensional database (IDB)*—the predicates calculated based on the EDB and the program rules. Semantically, a model of a Datalog program is a choice of IDB relations that, with the given EDB relations, makes all program rules true for all variable substitutions. Like Prolog, pure Datalog features *set semantics*, meaning that it does not distinguish between multiple instances of the same fact.

**Datalog$^{\mathrm{LB}}$.** The goal of Datalog$^{\mathrm{LB}}$ is to provide a generally useful, declarative way for expressing data structures, relations between data entities, sophisticated calculations, integrity constraints, and transactional processing. Datalog$^{\mathrm{LB}}$ is a type-safe variant of Datalog, based on incremental evaluation, with trigger-like functionality and support for dynamic updates, ability to declaratively specify functional dependencies, non-deterministic choice, stratified negation and aggregation, and meta-programming. Datalog$^{\mathrm{LB}}$ retains pure Datalog's set semantics, admitting at most one instance of any fact in program's database.

### 2.2 Constraint Handling Rules

**Syntax.** A CHR program is a finite set of rules that specify how multisets of *user-defined* constraints are solved based on the host language's *built-in* constraints (e.g. Prolog predicates). CHR rules are of the form:

$$\textit{label} \ @ \ \textit{Head} \left\{ \begin{array}{l} <=> \\ ==> \end{array} \right\} \textit{Guard} \mid \textit{Body}$$

The most general are *simpagation* rules of the form $H_1 \setminus H_2$ <=> $G \mid B$ where $H_1$ and $H_2$ are sequences of user-defined constraint terms (the *heads* of the rule), $G$ (the *guard*) is a sequence of built-in constraints and $B$ (the *body*) is a sequence of built-in and user-defined constraint terms. A rule specifies that when constraints in the store match $H_1$ and $H_2$ and the guard $G$ holds, the constraints that match $H_2$ can be *replaced* by the corresponding constraints in $B$. The literal `true` represents an empty sequence of constraint terms. The guard part, $G \mid$, may be omitted when $G = \textit{true}$.

A *simplification* rule, which has the form $H_2$ <=> $G \mid B$ can be represented by a simpagation rule `true` $\setminus H_2$ <=> $G \mid B$. Similarly, a *propagation* rule, which has the form $H_1$ ==> $G \mid B$, can be represented by a simpagation rule $H_1 \setminus$ `true` <=> $G \mid B$.

**Semantics.** CHR has a well-defined declarative as well as operational semantics [1, 11]. The declarative interpretation of a CHR program $P$ is given by the set of universally quantified formulas corresponding to the CHR rules, and an underlying consistent constraint theory.The constraint theory defines the meaning of host language constraints, the equality constraint '=', and the boolean atoms *true* and *false*.

The original operational semantics of CHR [11] is given in terms of a non-deterministic transition system. The evaluation of a program $P$ is a path through the transition system. The transitions are made when a constraint is added from the goal to the store,

```
:- constraints parent/2, ancestor/2, sibling/2.

parent   @ parent(X,Y) ==> ancestor(X,Y).
ancestor @ parent(X,Y), ancestor(Y,Z) ==> ancestor(X,Z).

sibling  @ parent(P,X), parent(P,Y) ==> X\==Y | sibling(X,Y).

setsem   @ sibling(X,Y) \ sibling(X,Y) <=> true.
```

**Table 1.** A CHR program for deductive database of family relations

or by firing any applicable program rule. The refined operational semantics [12], followed by most CHR implementations, defines a more deterministic transition system specifying, among others, the order in which rules are tried. The refined operational semantics is shown to be sound and to have better termination behavior than the original semantics.

*Example 1.* Table 1 lists a CHR program encoding a simple deductive database of family relations. The propagation rule `parent`, for each stored constraint matching `parent(X,Y)`, adds to the store an `ancestor(X,Y)` constraint; the propagation rule `ancestor`, for each stored pair of constraints matching `parent(X,Y)` and `ancestor(Y,Z)`, adds an `ancestor(X,Z)` constraint; similarly, the propagation rule `sibling`, for each pair of stored constraints that match `parent(P,X)` and `parent(P,Y)`, where X≠Y, adds a `sibling(X,Y)` entry to the constraint store. The simpagation rule `setsem`, in the presence of two identical stored constraints matching `sibling(X,Y)`, replaces one of these constraints (to the right of '\') with `true`, effectively removing the constraint from the store.

The evaluation of the program is triggered by a goal formed by a sequence of user-defined constraints. The propagation rules deduce the `ancestor` and `sibling` constraints implied by the given `parent` constraints, whereas the simpagation rule `setsem` removes duplicate occurrences of the `sibling` constraint.

## 3 Basic Translation Schema: CHR to Pure Datalog

In this section we analyze Constraint Handling Rules in the context of the characteristic properties of pure Datalog, and identify the class of CHR directly expressible as pure-Datalog programs.

### 3.1 CHR vs. pure Datalog

**Always-ground facts.** Evaluation of a query over a pure-Datalog program computes a fixed point according to a set of rules and a relational database expressed as a set of facts, and the groundness of the facts guarantees its termination. In the context of CHR, facts correspond to the contents of the constraint store. To ensure that all constraints stored

during the evaluation of a CHR program are ground, we require that (i) the program is *range restricted*, meaning that whenever a variable appears in a program rule's body, it also appears in the rule's head[1], and (ii) all queries issued to the program are ground.

**No function symbols.** Pure Datalog's restriction to 0-ary function symbols manifests in the context of CHR as two requirements: (i) that all constraint arguments in program rules are variables or constants, and (ii) that no functions (e.g., arithmetic) are evaluated in program rules' bodies. For programs in which function symbols appear as arguments only in the rules' heads, we can lift the requirement (i) by applying one of the flattening techniques introduced in [13]. Thus, this restriction admits to translation into pure Datalog all CHR programs as long as their flattened versions comply with all other requirements discussed in this section.

**Negation freedom.** Datalog's property of negation freedom allows adding new facts to a database, but does not allow removing any facts that are already there. Thus, at any step of the evaluation, new facts are derived based on all facts added in the previous steps. Furthermore, since we only add facts, and do so until no more facts are implied by the current facts set and program rules, the order in which the facts are derived (i.e., the order in which the program rules are applied) does not affect the final result of the evaluation, meaning that all pure-Datalog programs are confluent. On the other hand, CHR supports constraint removal by means of simplification, which enables writing non-confluent programs. In this section, when relating CHR to pure Datalog, we consider only the simplification-free subset CHR. In Section 4 we identify a class of programs with restricted simplification, which can be represented in Datalog$^{\mathrm{LB}}$, and in Section 5 we discuss issues around mapping to Datalog CHR with full-fledged simplification.

### 3.2 Translation schema

We now characterize the class of CHR programs amenable for direct translation into pure Datalog, and define the translation schema for this program class.

**Definition 1 (CHR$^{\delta}$ rule).** *A* CHR$^{\delta}$ rule *is a range-restricted CHR propagation rule, in which all arguments of the body constraints are terms of arity* $< 1$.

**Definition 2 (CHR$^{\delta}$ rule mapping).** *The mapping* $m_{\delta} : CHR^{\delta} \mapsto D$ *from CHR*$^{\delta}$ *rules to pure-Datalog clauses is defined as:*

$$m_{\delta}(\texttt{H ==> G | B}) \quad = \quad \texttt{B <- H,G.}$$

Because the semantics of Datalog does not distinguish duplicate facts, translating into Datalog CHR programs that place multiple instances of the same constraint in the store will change their behavior. In our previous work [14] we identified the class of *set-CHR* programs, for which the constraint store is always a set. Clearly, translation to Datalog is useful only for set-CHR programs. A common way to enforce set semantics in CHR is by enhancing the programs with simpagation rules of the form: `c \ c <=> true.` for every constraint symbol `c`, for which multiple constraint instances may be added to the store during the evaluation. Rules of this kind, which we

[1] this property coincides with the safety property of Datalog rules (see Section 2.1)

call *set-semantics rules*, explicitly remove duplicate constraints, and are often utilized in set-CHR programs. Guaranteed semantics of the output of our translation allows to omit the set-semantics rules from the input programs:

**Definition 3 (Set-semantics rule elimination).** *The elimination of CHR set-semantics rules $m_\sigma$ : CHR$^\delta \mapsto D$ is defined as:*

$$m_\sigma(\texttt{c} \setminus \texttt{c} \texttt{ <=> } \texttt{true}) \quad = \quad \textit{true}.$$

Recall that the set of predicates in a Datalog program is partitioned into the EDB and the IDB. Intuitively, EDB predicates are editable by the outside world and cannot be modified by the system, whereas IDB predicates are calculated by the system and cannot be edited by the outside world. In the context of CHR, the EDB are the constraints provided by queries, and the IDB are the constraints deduced by rule application.

*Example 2.* In the family database program in Table 1, the constraint symbol `parent` appears in the heads of all rules, and never in rule bodies. Thus, all instances of the constraint in the constraint store are those provided by the goal. Furthermore, `parent` is a premise for deducing all other constraints, as the presence of its instances in the store warrants applicability of all propagation rules. Clearly, this constraint represents an EDB predicate. By contrast, the constraint symbols `ancestor` and `sibling` appear in rule bodies, and so, the instances of these constraints are deduced by rule application. As such, the constraints represent the IDB predicates.

Even with a clean distinction between the constraint representation of the EDB and IDB in a CHR program, as in Example 2, nothing prevents posing the constraints representing the IDB in the queries, thus confusing these constraints with those representing the EDB. To avoid similar confusion in Datalog programs generated by our translation, we explicitly separate the constraints representing the IDB in program rules from their counterparts allowed in the queries by introducing an *EDB predicate* and an *EDB rule* for each constraint representing an IDB predicate in source CHR programs:

**Definition 4 (EDB predicate and EDB rule).** *The* EDB predicate $p_\epsilon$ *represents the IDB predicate $p$ in the EDB. The* EDB rule *for a predicate $p$, $r_\epsilon(p)$, maps the EDB predicate $p_\epsilon$ to its IDB counterpart:*

$$r_\epsilon(p) \quad = \quad p \texttt{ <- } p_\epsilon$$

**Definition 5 (Program translation).** *A pure-Datalog translation of a CHR program given by a set of CHR$^\delta$ rules and a set of set-semantics rules over a set of user-defined constraints, $P(C) = R_\delta \cup R_\sigma$, is a program $m_\pi(P)$ in which each constraint $c \in C$ representing an IDB predicate is associated with an EDB rule, each rule $r \in R_\delta$ is mapped to a pure-Datalog clause, and all set-semantics rules are eliminated: $m_\pi(P) = r_\epsilon(C) \cup m_\delta(R_\delta) \cup m_\sigma(R_\sigma)$.*

*Example 3.* The family database program in Table 1 consists of three CHR$^\delta$ rules (`parent`, `ancestor`, and `sibling`), and a set-semantics rule (`setsem`). The constraint `parent` represents the EDB predicate, whereas the constraints `ancestor` and `sibling` represent the IDB predicates. The pure-Datalog translation of the program is shown in Table 2, where lines $1, 2$ list the EDB rules, lines $4, 5, 7$ list direct mapping of the CHR$^\delta$ rules to pure Datalog, and the set-semantics rule has been eliminated.

```
ancestor(X,Y) <- ancestor_ε(X,Y).                                    1
sibling(X,Y) <- sibling_ε(X,Y).                                      2
                                                                     3
ancestor(X,Y) <- parent(X,Y).                                        4
ancestor(X,Z) <- parent(X,Y), ancestor(Y,Z).                         5
                                                                     6
sibling(X,Y) <- parent(P,X), parent(P,Y), X\=Y.                      7
```

**Table 2.** A pure-Datalog representation of the family database program

*Correctness* The CHR programs expressible in pure Datalog are confluent and their evaluation always terminates. The basic translation presented in this section is sound and complete w.r.t. the data sets deduced by the input and output programs.

**Theorem 1 (Soundness and Completeness).** *The constraint store resulting from the evaluation of a goal $Q$ over a CHR program $P = R_\delta \cup R_\sigma$ is equivalent to the set of facts deduced by the pure-Datalog translation of $P$, $m_\pi(P)$, for a set of EDB facts $F$ such that $F = Q$.*

Theorem 1 holds based on the pure-Datalog semantics and the logical reading of CHR.

## 4 Extended Translation Schema: CHR to Datalog$^{\mathrm{LB}}$

In Section 3.2 we defined a mapping from CHR to pure Datalog. The mapping is straightforward, and the subclass of CHR programs amenable for the mapping are guaranteed to have the properties of pure Datalog programs such as termination and confluence. This subclass of CHR, however, is very small and leaves out many practical programs. In this section we propose three extensions to the basic translation schema, which accommodate features common in CHR, but not standard to pure Datalog. The extensions, facilitated by the properties of the Datalog$^{\mathrm{LB}}$ system underlying our translation, and by a simple CHR program transformation, still yield a translation schema that guarantees well-behavedness of the input programs.

### 4.1 Restricted simplification

Pure Datalog's requirement of negation freedom is perhaps the most prohibitive restriction of the language, and numerous approaches have been taken towards relaxing it. For example, many Datalog systems allow programs with *stratified negation*, i.e., programs in which all instances of any predicate appearing in negated subgoals are computed before the predicate is used with negation.

The negation-freedom requirement is very restrictive also in the context of our translation. As argued in Section 3.1, CHR implements negation by means of simplification. Hence, the requirement bans all simplification from the basic translation schema, which severely limits the schema's applicability, as simplification rules are dominant in most

CHR programs. In this section we identify a subclass of CHR with *stratified simplification*, for which we can lift this restriction.

Intuitively, a program is simplification stratified if it is never the case that the simplification of a constraint triggers further propagation. Our formal definition of simplification-stratified programs is based on the notion of *constraint dependency graph*:

**Definition 6 (Constraint dependency graph).** *A* constraint dependency graph *for a CHR program $P$ is a graph $G = \langle N, E \rangle$ with a set of nodes $N$ and a set of edges $E$, in which the set of nodes is the set of user-defined constraints, and there is an edge on a rule $r \in P$, denoted $e(r)$, from a source node $c_s$ to a target node $c_t$, if the constraint represented by $c_s$ appears in the head of the rule $r$, and the constraint represented by $c_t$ appears in the body of $r$.*

**Definition 7 (Negative edge and positive edge).** *A* negative edge *in a constraint dependency graph is an edge $e(r)$ from a source node $c_s$ such that $r$ is a simplification rule, or $r$ is a simpagation rule and its application removes the constraint $c_s$. A* positive edge *is an edge that is not negative.*

**Definition 8 (Simplification-stratified program).** *A* simplification stratified program *is a CHR program such that for all nodes in its constraint dependency graph it holds that if a node has an outgoing edge, then all its incoming edges are positive.*

The evaluation of a simplification-stratified program can be split into two conceptual steps, with a propagation step—computing all constraints (solution candidates) implied by a given goal—followed by a simplification step—applying the solution selection criteria which identify the actual solution. This property enables translation of simplification-stratified programs into negation-stratified Datalog$^{\mathrm{LB}}$ programs defining the following sequence of operations:

1. based on the EDB facts and rules, derive the solution candidates (IDB facts)
2. identify the candidates that do not satisfy the program's solution selection criteria
3. determine the solution as the set of candidates not ruled out by the selection criteria.

We formally define the extended translation schema in Section 4.4. Here we illustrate the extension with an example CHR program and its Datalog$^{\mathrm{LB}}$ counterpart.

*Example 4.* Table 3(a) lists a simplification-stratified program that, given a set of numbers, finds its smallest element. The program rule iterates over the elements of the set, stored as individual constraints, and, upon finding one that is greater than some other set element, removes its representation from the constraint store. The Datalog$^{\mathrm{LB}}$ representation of the program is listed in Table 3(b). The program directly implements the three-step sequence outlined above. Line 1 identifies all set numbers as potential minimum elements, line 2 compares the numbers pairwise and adds all that are greater than some other number in the set to the predicate $\texttt{min}_{\neg}$, whereas line 3 identifies the actual set minimum as the element defined in `min` but not in $\texttt{min}_{\neg}$.

Many of the early formulations of CHR, e.g., [1,11], considered a simpagation rule of the form $\texttt{R@H}_1\backslash\texttt{H}_2\texttt{<=>G|B}$ a syntactic abbreviation—and thus a semantic equivalent—of a simplification rule of the form $\texttt{R'@H}_1\texttt{,H}_2\texttt{<=>G|H}_1\texttt{,B}$. The notion of simplification stratification enables the following observation about this relationship. Given a

```
:- constraints min/1.

min(I) \ min(J) <=> J>=I | true.
```

(a)

```
min(I) <- min_ϵ(I).                                    1
min_¬(J) <- min(I), min(J), I<J.                       2
min_ω(I) <- min(I), !min_¬(I).                         3
```

(b)

**Table 3.** A CHR program finding the smallest number in a set (a), and its representation in Datalog$^{\mathrm{LB}}$ (b)

simplification-stratified program $P$ comprising a rule R, by replacing the rule with its "equivalent" R′, we add to $P$'s constraint dependency graph a self-loop (i.e., both incoming and outgoing) edge on each node representing a constraint in $\mathtt{H}_1$, and a negative incoming edge from each node representing a constraint in $\mathtt{H}_2$ to each node representing a constraint in $\mathtt{H}_1$. Clearly, the resulting program is not simplification stratified, meaning that, in general, the two kinds of rules are not equivalent.

### 4.2 Restricted function symbols

Datalog's termination guarantee follows from the fact that the interpretation of every predicate is a finite relation: for an $n$-ary predicate $P$, $P \subset U_1 \times U_2 \ldots \times U_n$ where each $U_i$ is a finite Herbrand universe of 0-ary function symbols. Operationally, this enables evaluation of Datalog programs by a search through a finite number of interpretations. Extending Datalog with function symbols makes the Herbrand universe of terms infinite, and introduces the possibility of the logic engine exhaustively searching/enumerating an infinite space of solutions. Thus, pure Datalog—and our basic translation schema—disallow the use of function symbols of arity $> 0$. On the other hand, artithmetic functions, for instance, are very common in CHR, and hence relaxing this restriction may considerably expand the class of programs amenable for our translation.

Datalog$^{\mathrm{LB}}$ facilitates declaration of predicates as *functions* rather than just as relations, by specifying their domains and codomains: $p(d_1, \ldots, d_n, t_1, \ldots t_n)$. Such predicates are interpreted as functions: $d_1 \times \ldots \times d_n \mapsto t_1 \times \ldots \times t_n$. The Datalog$^{\mathrm{LB}}$'s type system verifies that the universes for all $d_i$ are finite. With a finite domain, even if the interpretation of any $t_i$ is infinite, the function itself is guaranteed to be finite as well.We exploit this feature of Datalog$^{\mathrm{LB}}$ to allow the use of infinite-domain functions in a way that preserves termination guarantees of pure Datalog.

*Example 5.* Table 4(a) lists a CHR program computing a distance from an arbitrary node in a tree to the tree's root (the depth of the node). The rule `root` sets the depth of the tree's root node to 0. The rule `node` repeatedly descends from a node to the node's child, and updates the depth counter. Even though the update applies an infinite-domain

```
:- constraints root/1, edge/2, depth/2.

root @ root(N) ==> depth(N,0).
node @ edge(N1,N2), depth(N1,D1) ==> D2 is D1+1, depth(N2,D2).
```

(a)

```
depth(N,0) <- root(N).
depth(N2,D2) <- edge(N1,N2), depth(N1,D1), D2 = D1+1.
```

(b)

**Table 4.** A CHR program calculating depth of a tree node (a), and its representation in Datalog$^{\mathrm{LB}}$ (b)

function ('+') to the counter value, the program is well-behaved. This is because the value of the counter functionally depends on the node, and so, the number of the increment operations is bound by the number of the nodes in the tree. The program translates directly to Datalog$^{\mathrm{LB}}$, and its representation is listed in Table 4(b).

### 4.3 Restricted unboundedness

The requirement that all pure-Datalog facts are ground complies with the original purpose of the language, which was to facilitate specification of database queries and (recursive) views, but makes pure Datalog inapplicable to problems that involve *top-down recursion* i.e., recursion through value-computing (in a broad sense) predicates. Since such problems are easily, and commonly, represented in CHR, searching for ways to relax this restriction seems worthwhile.

*Example 6.* Table 5(a) lists a CHR program encoding the naive union-find algorithm. The algorithm defines a forest of disjoint sets and two operations on its elements: *find* to determine which tree in the forest contains a given node, and *union* to merge two trees into one. A tree is represented by its root node. In the program, the constraints `root` and `->` capture the structure of the forest, whereas the constraints `make`, `union`, `find`, and `link` define the operations. The rule `make` creates a new tree with a single node, and designates that node as the tree's root. The rule `union`, given two nodes, merges the trees containing these nodes by finding the root of each tree and linking the two roots together. The rules `findNode` and `findRoot` repeatedly advance from a given node to its parent until reaching a root. The rules `linkEq` and `link` create a new tree by merging two existing root nodes, and designate one of these nodes as the tree's root.

The program in Table 5(a) defines recursive value computation by means of the `find` constraint which, given a node in a tree, returns the tree's root. The constraint is activated by the body of the `union` rule, with its first argument bound to the name of the node, and its second argument unbound. Activation of the constraint triggers either

```
:- constraints make/1, find/2, union/2, (->)/2, link/2, root/1.

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(B) \ find(B,X) <=> X=B.

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

(a)

```
:- constraints make/1, find/2, union/2, (->)/2, link/2, root/1,
               eq/2.

refl     @ eq(X,X) ==> true.
symm     @ eq(X,Y) ==> eq(Y,X).
trans    @ eq(X,Y), eq(Y,Z) ==> eq(X,Z).

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(B) \ find(B,X) <=> eq(X,B).

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

(b)

**Table 5.** The naive union-find algorithm in traditional CHR encoding (a), and with user-defined unification (b)

the `findNode` or the `findRoot` rule, and only the latter unifies the second argument with the name of the tree's root node. Admitting unbound constraint arguments in a rule's body violates the range-restrictedness requirement of CHR$^{\delta}$, and, in the context of Datalog, leads to generation of non-ground facts, thus enforcing subgoal ordering. Hence, programs with top-down recursion cannot be represented in pure Datalog.

The translation of CHR programs with top-down recursion into Datalog$^{\mathrm{LB}}$ is enabled by a simple program transformation. The goal of the transformation is to replace the built-in constraint responsible for the delayed binding of the value argument with a user-defined constraint that can be added to the store (thus simulating the binding) at

any time of the evaluation. Transformed CHR programs are amenable for translation using the extended schema formalized in Section 4.4.

*Example 7.* Table 5(b) lists the transformed union-find program which defines a new constraint, `eq`, representing unification. The constraint is used in the body of the `findRoot` rule to reflect the binding of the variable to the name of the root node. Three new rules, `refl`, `symm` and `trans`, define the properties of the `eq` constraint.

### 4.4 Translation schema

The basic translation schema from Section 3.2 facilitates pure-Datalog representation of CHR programs required (i) to contain only range-restricted propagation rules free from function symbols, and (ii) to be evaluated only for ground queries. Clearly, these requirements demarcate a very small, and not very useful, subclass of CHR. In this section we define an extended translation schema for mapping a more interesting subclass of CHR into Datalog$^{\mathrm{LB}}$, by relaxing the basic schema's restrictions on program syntax in the following ways:

1. admit programs with simplification rules (governed by stratified simplification, Section 4.1)
2. admit rules with function symbols and local variables (governed by functional dependency, Section 4.2)
3. admit non-ground queries over programs (governed by user-defined built-ins, Section 4.3)

Hence, we generalize the notion of a rule amenable for translation:

**Definition 9 (CHR$^{\chi}$ rule).** *A* CHR$^{\chi}$ rule *is a CHR rule*

$$H_1 \setminus H_2 \iff G \mid A, B$$

*where* $A$ *is a (possibly empty) sequence of arithmetic constraints;* $B$ *is a sequence of user-defined and built-in (non-arithmetic) constraints; and for every constraint* $c \in A \cup B$*, each variable argument of* $c$ *either appears in vars*$(H_1 \cup H_2)$*, or functionally depends on some variable in vars*$(H_1 \cup H_2)$*.*

Consequently, we generalize the definition of rule mapping:

**Definition 10 (Rule mapping $m_{\chi}$).** *The mapping* $m_{\chi} : CHR^{\chi} \mapsto D$ *from CHR*$^{\chi}$ *rules to Datalog*$^{\mathrm{LB}}$ *clauses is defined as:*

$$m_{\chi}(H_1 \setminus H_2 \iff G \mid A, B) \quad = \quad B, H_{2\neg} \text{ :- } H_1, H_2, G, A$$

*where the predicate* $H_{2\neg}$ *denotes a sequence of the negated versions of all constraint symbols appearing in the head* $H_2$*:* $H_{2\neg} = \{c_{\neg} \mid c \in H_2\}$*.*

The predicate $H_{2\neg}$ simulates constraint removal. Since Datalog$^{\mathrm{LB}}$ does not support explicit removal of data, we denote removal of an instance of an IDB predicate `p` from a program's database by adding a corresponding fact to $p_{\neg}$. After completed evaluation, the database contains a set of facts added to `p`, in `p` itself, and a set of facts (marked as) removed from `p`, in $p_{\neg}$. We identify the facts that are actually in the database (i.e., are not marked as removed), by means of an *output predicate* and an *output rule*:

**Definition 11 (Output predicate and output rule).** *The* output predicate *for an IDB predicate $p$, $p_\omega$, represents the actual definition of $p$ in a program's database. The* output rule *for $p$, $r_\omega(p)$, defines the output predicate $p_\omega$ by selecting from the database the facts of $p$ that have been added, but not removed, during the evaluation:*

$$r_\omega(p) \quad = \quad p_\omega \texttt{ <- } p\texttt{, !}p_\neg\texttt{.}$$

The set-semantics rules elimination, as well as the EDB predicates and EDB rules are preserved from the basic translation schema.

**Definition 12 (Program translation).** *A Datalog$^{\mathrm{LB}}$ translation of a CHR program given by a set of CHR$^\chi$ rules and a set of set-semantics rules over a set of user-defined constraints, $P(C) = R_\chi \cup R_\sigma$, is a program $m_\pi(P)$ in which each constraint $c \in C$ representing an IDB predicate is associated with an EDB rule and an output rule, each rule $r \in R_\chi$ is mapped to a Datalog$^{\mathrm{LB}}$ clause, and all set-semantics rules are eliminated: $m_\pi(P) = r_\epsilon(C) \cup r_\omega(C) \cup m_\chi(R_\chi) \cup m_\sigma(R_\sigma)$.*

*Correctness* The extended translation schema generates Datalog$^{\mathrm{LB}}$ programs which are operationally equivalent to the source CHR programs, and preserve well-behavedness properties of pure Datalog.

## 5 Full-fledged simplification

In Section 4.4 we defined a translation schema facilitating Datalog$^{\mathrm{LB}}$ representation of an interesting, but restricted, class of CHR programs. In this section we discuss one of the main challenges of translating into Datalog the unrestricted CHR.

The schema from Section 4.4 admits for translation into Datalog$^{\mathrm{LB}}$ CHR programs with stratified simplification, in which all constraint removals may be performed in a single (and final) evaluation step. In CHR programs that are not simplification stratified, simplification of constraints is interleaved with propagation, and it is possible that a constraint `c` added to the store at some point of the evaluation, is later removed, and then added again. Such behavior cannot be represented in Datalog$^{\mathrm{LB}}$ simply by means of a $\texttt{c}_\neg$ predicate as before, because this can capture at most one addition and removal of any given fact[2]. To reflect subsequent additions and removals, we need to extend the Datalog$^{\mathrm{LB}}$ predicate representing the constraint `c` with a time dimension, which will allow to keep track of the contents of the predicate at every point of program evaluation. We have implemented this approach by means of time stamps, or *steps*, added to each predicate in the translated program. With steps, a CHR simplification rule:

```
H <=> G | B
```

translates into a Datalog$^{\mathrm{LB}}$ rule:

```
B(SN), H(SN), H¬(SP) <- H(SP), G, next_step(SP,SN)
```

[2] recall that set semantics treats identical facts (constraint instances) as the same fact (constraint)

where `SP` and `SN` correspond to, respectively, the previous and the next step, and the predicate `next_step` advances from one step to the other. Simulating the time dimension in Datalog$^{\mathrm{LB}}$ takes away much of the programs' declarativeness, and leads to a significant increase in their state space. Furthermore, programs with time dimension are guaranteed to terminate only if the number of the steps taken is finite. To ensure this requires pre-allocating the steps at the beginning of the evaluation. If the number of steps necessary to fully process a problem cannot be determined statically, this may result in the evaluation terminating early, before reaching the result. Given these shortcomings, the benefit of pursuing this direction of our approach is yet to be determined.

## 6 Discussion and Related Work

In this paper, by establishing a relationship between CHR and two variants of Datalog, we have enabled reasoning about the logical reading of CHR programs in terms of the declarative and well-studied Datalog logic. Our basic translation schema shows a clean correspondence between pure Datalog and a small subclass of CHR. The programs in this CHR subclass are guaranteed to hold strong termination and confluence properties of pure Datalog. The extended translation schema significantly augments that basic subclass of CHR, at the same time preserving well-behavedness guarantees for the admitted programs. We expect to continue our investigation of this relationship, possibly enhancing it with the consideration of other relevant formalisms.

Connections between CHR and other formalisms have been studied before [1, 15–17]. First, a representation of CHR rules as universally quantified formulas in first-order predicate logic, together with a built-in constraint theory of the host language, defined a program's classical declarative semantics [1]. This approach turned out not always accurate (e.g., for programs with multiset semantics or procedural use of CHR[3]), opening the way to alternative interpretations. For instance, mapping CHR to intuitionistic linear logic [15] proved better suited for providing declarative semantics to programs with dynamic updates relying on non-deterministic committed choice. Even more accurate interpretation of procedural applications of CHR has been accomplished by transforming CHR programs into transaction logic [16], which uses a time-based dimension to reason at the level of individual derivation steps rather than only at the level of the final results. Our approach, relating CHR to (extended) Datalog, restores to the search for a declarative interpretation of CHR. Even though both pure Datalog and Datalog$^{\mathrm{LB}}$ lack the time-based dimension that allows to express full-fledged simplification, our translation enables declarative reasoning about a substantial subclass of CHR with restricted simplification, and we can simulate full-fledged simplification by extending the programs with a notion of pre-allocated evaluation steps.

CHR has been also related to (colored) Petri nets (CPN) [17]. The approach exploits a positive, range-restricted, ground subset of CHR, and proposes CPN-based analysis of concurrency properties of programs in this subset to facilitate parallelizing their execution. Our extended translation schema considers a larger subset of CHR (we relax both the range-restrictedness and groundness requirements), for which it guarantees confluence, thus opening the programs in this set to parallel processing.

[3] meaning, the use of CHR to express temporal, rather than purely logical, consequence

The connections between CHR and Datalog have been explored in the context of CHR$^\vee$ [18], an extension of CHR with disjunction, which facilitates CHR-based representation of (disjunctive) deductive databases. With support for mixing top-down and bottom-up programming paradigms, and admitting existentially quantified variables in rule bodies, the approach is an interesting complement to our work, and further studies of the relationship between the two seem worthwhile.

## References


1. Frühwirth, T.: Theory and Practice of Constraint Handling Rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37**(1-3) (1998) 95–138
2. CHR. http://www.cs.kuleuven.ac.be/∼dtai/projects/CHR/
3. Lloyd, J.W.: Foundations of Logic Programming. Springer (1984)
4. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
5. Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: Symp. on Principles of Database Systems (PODS). (2005)
6. Loo, B., Condie, T., Garofalakis, M., Gay, D., Hellerstein, J., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking: Language, execution and optimization. In: Intl. Conf. on Management of Data (SIGMOD). (2006)
7. Li, N., Mitchell, J.: Datalog with constraints: A foundation for trust-management languages. In: Symp. on Practical Aspects of Declarative Languages (PADL). (2003)
8. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Logic **7**(3) (2006) 499–562
9. Ashley-Rollman, M.P., Rosa, M.D., Srinivasa, S.S., Pillai, P., Goldstein, S.C., Campbell, J.D.: Declarative programming for modular robots. In: Workshop on Self-Reconfigurable Robots/Systems and Applications. (2007)
10. White, W., Demers, A., Koch, C., Gehrke, J., Rajagopalan, R.: Scaling games to epic proportions. In: Intl. Conf. on Management of Data (SIGMOD). (2007)
11. Abdennadher, S.: Operational Semantics and Confluence of Constraint Propagation Rules. In: Intl. Conf. on Principles and Practice of Constraint Programming (CP). (1997)
12. Duck, G.J., Stuckey, P.J., de la Banda, M.J.G., Holzbaur, C.: The Refined Operational Semantics of Constraint Handling Rules. In: Intl. Conf. on Logic Programming (ICLP). (2004)
13. Sarna-Starosta, B., Schrijvers, T.: Indexing Techniques for CHR based on Program Transformation. Technical Report CW 500, Department of Computer Science, K.U.Leuven (2007)
14. Sarna-Starosta, B., Ramakrishnan, C.: Compiling Constraint Handling Rules for Efficient Tabled Evaluation. In: Symp. on Practical Aspects of Declarative Languages (PADL). (2007)
15. Betz, H., Frühwirth, T.: A linear-logic semantics for Constraint Handling Rules. In: Intl. Conf. on Principles and Practice of Constraint Programming (CP). (2005)
16. Meister, M., Djelloul, K., Robin, J.: A unified semantics for Constraint Handling Rules in transaction logic. In: Intl. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR). (2007)
17. Betz, H.: Relating coloured Petri nets to Constraint Handling Rules. In: Workshop on Constraint Handling Rules (CHR). (2007)
18. Abdennadher, S., Schütz, H.: CHR$^\vee$: A flexible query language. In: Intl. Conf. on Flexible Query Answering Systems (FQAS). (1998)

# Generalized CHR Machines

Jon Sneyers [⋆1] and Thom Frühwirth[2]

[1] K.U. Leuven, Belgium
jon.sneyers@cs.kuleuven.be
[2] University of Ulm, Germany
thom.fruehwirth@uni-ulm.de

**Abstract.** Constraint Handling Rules (CHR) is a high-level rule-based programming language. In [1, 2], a model of computation based on the operational semantics of CHR is introduced, called the CHR machine. The CHR machine was used to prove a complexity-wise completeness result for the CHR language and its implementations. In this paper, we investigate three generalizations of CHR machines: CHR machines with an instantiated operational semantics, non-deterministic CHR machines, and self-modifying CHR machines.

## 1 Introduction

Constraint Handling Rules is a high-level language extension based on multi-headed committed-choice rules [3–5]. The abstract operational semantics $\omega_t$ of CHR is very non-deterministic. Since rule applications are committed-choice — CHR has no built-in search mechanisms like backtracking — confluence of a CHR program is a crucial property.

In earlier work [1, 2] we have shown a complexity-wise completeness result for CHR: everything (every RAM-machine program) can be implemented in CHR (in a confluent way), and there are CHR systems which execute the resulting CHR program with the right time and space complexity. Recently, Di Giusto et al. have shown that even single-headed CHR rules suffice to implement a Turing-equivalent Minsky machine [6], although they argue that in terms of expressive power, multiple heads are still needed.

In the approach of [1, 2], a theoretical "CHR machine" is defined, which performs one $\omega_t$ transition in every step. In this paper we propose several more general definitions for CHR machines. This is theoretical work, investigating how classical (variants of) models of computation can be transferred to the setting of CHR machines. Section 2 recaptures some of the definitions of [2]. The particular kind of CHR machines used in [1, 2] are *deterministic abstract CHR machines*, in the terminology of this paper. Section 3 introduces the concept of *strategy classes*, which formalizes instantiations of the $\omega_t$ operational semantics.

⋆ Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). This research was performed in large part while Jon Sneyers was visiting the University of Ulm.

We also define and discuss a more general notion of confluence. Then, in Section 4, we consider CHR machines with an instantiated operational semantics (for example the refined operational semantics [7]). Finally, in Section 5, we define non-deterministic CHR machines, and in Section 6, we define CHR machines with a stored program, also known as *self-modifying* CHR machines.

## 2 Deterministic Abstract CHR Machines

We assume the reader to be familiar with CHR. We will use the same notation as in [2]. That is, we denote the host language with $\mathcal{H}$, the built-in constraint theory with $\mathcal{D}_\mathcal{H}$, the set of queries for a program $\mathcal{P}$ with $\mathcal{G}_\mathcal{P}^\mathcal{H}$, the abstract (or theoretical) operational semantics with $\omega_t$, its execution states with $\sigma, \sigma_0, \sigma_1, \ldots$ and the set of all execution states with $\Sigma^{\text{CHR}}$, the $\omega_t$ transition relation with $\rightarrowtail_\mathcal{P}$ and its transitive closure with $\rightarrowtail_\mathcal{P}^*$. We use $\bar{\exists}\phi$ as a shorthand for existentially quantifying over all free variables of $\phi$, and $\bar{\exists}_V\phi$ to existentially quantify over all free variables of $\phi$ except those of $V$. Finally, we use $\in\!\!\in$ to denote "is an element of an element of": $x \in\!\!\in X \Leftrightarrow \exists A \in X : x \in A$.

### 2.1 Derivations

**Definition 1.** *Given an* initial goal *(or* query*)* $\mathbb{G} \in \mathcal{G}_\mathcal{P}^\mathcal{H}$*, the corresponding* initial state *is defined as* $\mathit{initstate}(\mathbb{G}) = \langle \mathbb{G}, \emptyset, \mathit{true}, \emptyset \rangle_1$*. We denote the set of all initial states by* $\Sigma^{\mathit{init}} \subset \Sigma^{\text{CHR}}$*.*

**Definition 2.** *A* final state $\sigma_f = \langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ *is an execution state for which no transition applies:* $\neg\exists\sigma \in \Sigma^{\text{CHR}} : \sigma_f \rightarrowtail_\mathcal{P} \sigma$*. In a* failure *state, the underlying solver* $\mathcal{H}$ *can prove* $\mathcal{D}_\mathcal{H} \models \neg\bar{\exists}\mathbb{B}$ *— such states are always final. A* successful *final state is a final state that is not a failure state, i.e.* $\mathcal{D}_\mathcal{H} \models \bar{\exists}\mathbb{B}$*. The set of final states is denoted by* $\Sigma^{\mathit{final}} \subset \Sigma^{\text{CHR}}$*.*

**Definition 3.** *Given a CHR program* $\mathcal{P}$*, a* finite derivation $d$ *is a finite sequence* $[\sigma_0, \sigma_1, \ldots, \sigma_n]$ *of states where* $\sigma_0 \in \Sigma^{\mathit{init}}$*,* $\sigma_n \in \Sigma^{\mathit{final}}$*, and* $\sigma_i \rightarrowtail_\mathcal{P} \sigma_{i+1}$ *for* $0 \leq i < n$*. If* $\sigma_n$ *is a failure state, we say* $d$ *has* failed*, otherwise* $d$ *is a* successful *derivation.*

**Definition 4.** *An* infinite derivation $d_\infty$ *is an infinite sequence* $\sigma_0, \sigma_1, \ldots$ *of states where* $\sigma_0 \in \Sigma^{\mathit{init}}$ *and* $\sigma_i \rightarrowtail_\mathcal{P} \sigma_{i+1}$ *for* $i \in \mathbb{N}$*.*

We use $\#d$ to denote the length of a derivation: the length of a finite derivation is the number of transitions in the sequence; the length of an infinite derivation is $\infty$. A set of (finite or infinite) derivations is denoted by $\Delta$. The set of all derivations in $\Delta$ that start with $\mathit{initstate}(\mathbb{G})$ is denoted by $\Delta|_\mathbb{G}$. We use $\Delta_{\omega_t}^\mathcal{H}(\mathcal{P})$ to denote the set of all derivations (in the $\omega_t$ semantics) for a given CHR program $\mathcal{P}$ and host language $\mathcal{H}$. We now define the relation $\rightsquigarrow_\Delta$: $\Sigma^{\mathit{init}} \rightarrow \Sigma^{\text{CHR}} \cup \{\infty\}$:

**Definition 5.** *State* $\sigma_n$ *is a* $\Delta$-output *of* $\sigma_0$ *if* $[\sigma_0, \ldots, \sigma_n] \in \Delta$*. We say* $\sigma_0$ $\Delta$-outputs $\sigma_n$ *and write* $\sigma_0 \rightsquigarrow_\Delta \sigma_n$*. If* $\Delta$ *contains an infinite derivation starting with* $\sigma_0$*, we say* $\sigma_0$ *has a non-terminating derivation, denoted as* $\sigma_0 \rightsquigarrow_\Delta \infty$*.*

**Definition 6.** *The CHR program $\mathcal{P}$ is* $\Delta$-deterministic *for input $I \subseteq \mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$ if the restriction of $\rightsquigarrow_{\Delta}$ to initstate$[I]$ is a function and $\forall i \in I, d \in \Delta|_i$ : if $d$ is a successful derivation, then $\forall d' \in \Delta|_i : \#d = \#d'$.*

In other words, a program is $\Delta$-deterministic if all derivations starting from a given input have the same result and all successful ones have the same length. Note that if a program $\mathcal{P}$ is $\Delta_{\omega_t}^{\mathcal{H}}(\mathcal{P})$-deterministic for all input $\mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$, it is also observable confluent [8]. The notion of observable confluence does not require derivations to have the same length.

### 2.2 Deterministic Abstract CHR Machines

We now define a class of CHR machines, which corresponds to the definition given in [2]. This class of CHR machines is somewhat restricted since it only allows $\Delta_{\omega_t}^{\mathcal{H}}$-deterministic CHR programs. In Section 4 we will allow more general CHR machines.

**Definition 7.** *A* deterministic abstract CHR machine *is a tuple $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{WG})$. The host language $\mathcal{H}$ defines a built-in constraint theory $\mathcal{D}_{\mathcal{H}}$, $\mathcal{P}$ is a CHR program, and $\mathcal{WG} \subseteq \mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$ is a set of* valid goals*, such that $\mathcal{P}$ is a $\Delta_{\omega_t}^{\mathcal{H}}$-deterministic CHR program for input $\mathcal{WG}$. The machine takes an input query $\mathbb{G} \in \mathcal{WG}$ and executes a derivation $d \in \Delta_{\omega_t}^{\mathcal{H}}|_{\mathbb{G}}$.*

**Terminology.** If the derivation $d$ for $\mathbb{G}$ is finite, we say the machine *terminates* with *output state* $\mathcal{M}(\mathbb{G}) = \langle \mathbb{G}', \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ which is the last state of $d$. The machine *accepts* the input $\mathbb{G}$ if $d$ is a successful derivation and *rejects* $\mathbb{G}$ if $d$ is a failed derivation. If $d$ is an infinite derivation, we say the machine *does not terminate*. A *CHR(X) machine* is a CHR machine for which the host language $\mathcal{H} = X$. We use $\Phi$ to denote *no host language*: the built-in constraint theory $\mathcal{D}_{\Phi}$ defines only the basic constraints *true* and *fail*, and syntactic equality and inequality (only to be used as an *ask*-constraint). This implies that the **Solve** transition can only be used once (to add *fail*). The only data types are logical variables (that will not be bound) and constants. A *CHR-only machine* is a CHR($\Phi$) machine.

**Definition 8.** *A* sufficiently strong *host language $\mathcal{H}$ is a host language whose built-in constraint theory $\mathcal{D}_{\mathcal{H}}$ defines at least true, fail,* `==` *and* `\==`*, the integer numbers and the arithmetic operations for addition, subtraction, multiplication and integer division.*

Clearly, most host languages are sufficiently strong. Prolog for instance defines *true*, *fail*, `==` and `\==`, and allows arithmetic using the built-in `is/2`. In CHR program listings where the host language is assumed to be sufficiently strong, we will use a slightly abbreviated notation. For example, if **c**/1 is a CHR constraint, we write expressions like "**c**(N+1)": a host language independent notation that is equivalent to "M `is` N+1, **c**(M)" for CHR(Prolog), to "**c**(intUtil.add(N,1))" for CHR(Java), etc.

```
r1 @ delta(Q,S,Q2,S2,left), adj(LC,C)  \ state(Q),  cell(C,S),  head(C)
        <=> LC \== null |               state(Q2), cell(C,S2), head(LC).
r2 @ delta(Q,S,Q2,S2,right), adj(C,RC) \ state(Q),  cell(C,S),  head(C)
        <=> RC \== null |               state(Q2), cell(C,S2), head(RC).
r3 @ delta(Q,S,Q2,S2,left)   \ adj(null,C), state(Q),  cell(C,S),  head(C)
   <=> cell(LC,b), adj(null,LC), adj(LC,C), state(Q2), cell(C,S2), head(LC).
r4 @ delta(Q,S,Q2,S2,right)  \ adj(C,null), state(Q),  cell(C,S),  head(C)
   <=> cell(RC,b), adj(C,RC), adj(RC,null,  state(Q2), cell(C,S2), head(RC).
fail @ nodelta(Q,S), rejecting(Q), state(Q), cell(C,S), head(C) <=> fail.
```

**Fig. 1.** The CHR program TMSIM, a Turing machine simulator.

### 2.3 Computational Power of CHR Machines

We assume the reader to be familiar with Turing machines; we refer to [2] for a more detailed exposition.

A model of computation is called *Turing-complete* if it has the same computational power as Turing machines: every Turing Machine can be simulated in the model and every program of the model can be simulated on a Turing machine. Consider the CHR program TMSIM shown in Figure 1 and the corresponding CHR-only machine $\mathcal{M}_{\mathbb{TM}} = (\Phi, \mathsf{TMSIM}, \mathcal{VG}_{\mathbb{TM}})$. The program simulates Turing machines. Intuitively, the meaning of the constraints of TMSIM is as follows:

**delta/5** encodes the transition function (the Turing machine program) in the obvious way: the first two arguments are inputs, the last three are outputs;
**nodelta/2** encodes the domain on which $\delta$ is undefined;
**rejecting/1** encodes the set of non-accepting final states;
**state/1** contains the current state;
**head/1** contains the identifier of the cell under the head;
**cell/2** represents a tape cell. The first argument is the unique identifier of the cell. The second argument is the symbol in the cell.
**adj/2** encodes the order of the tape cells. The constraint $\mathbf{adj}(A, B)$ should be read: "the right neighbor of the tape cell with identifier $A$ is the tape cell with identifier $B$". The special cell identifier `null` is used to refer to a not yet instantiated cell. The rules $r_3$ and $r_4$ extend the tape as needed.

The set of valid goals $\mathcal{VG}_{\mathbb{TM}}$ corresponds to the goals that represent a valid Turing machine and a correctly represented tape; it is defined formally in [2].

A simulation of the execution of a Turing machine $M$ proceeds as follows. The tape input is encoded as $\mathbf{cell}/2$ constraints and $\mathbf{adj}/2$ constraints. The identifier of the cell to the left of the left-most input symbol is set to `null` and similarly for the cell to the right of the right-most input symbol. The transition function $\delta$ of $M$ is encoded in multiple $\mathbf{delta}/5$ constraints. All these constraints are combined in the initial query together with the constraint $\mathbf{state}(q_0)$ where $q_0$ is the initial state of $M$ and the constraint $\mathbf{head}(c_1)$ where $c_1$ is the identifier of the cell representing the left-most input symbol. Every rule application of the first four rules of TMSIM corresponds directly to a Turing machine transition.

If no more (Turing machine) transitions can be made, the last rule is applicable if the current state is non-accepting. In that case, the built-in constraint *fail* is added, which leads to a failure state. If the Turing machine ends in an accepting final state, the CHR program ends in a successful final state.

The program TMSIM can easily be rewritten to use only simplification rules. A Turing machine simulator can also be written using only propagation rules, or using only single-headed simplification rules, or with only guardless rules.

### 2.4 Time and Space Complexity

We define the time complexity of a CHR machine in an obvious way:

**Definition 9.** *Given a CHR machine $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{VG})$, the function $chrtime_{\mathcal{M}}$ returns the derivation length, given a valid goal:*

$$chrtime_{\mathcal{M}} : \mathcal{VG} \to \mathbb{N} : \mathbb{G} \mapsto \max\{\#d \mid d \in \Delta^{\mathcal{H}}_{\omega_t}|_{\mathbb{G}}\}.$$

**Definition 10.** *Given a CHR machine $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{VG})$ and assuming that host language constraints of $\mathcal{H}$ take constant time, the (worst-case) time complexity function* $\textsc{chrtime}_{\mathcal{M}}$ *is defined as follows:*

$$\textsc{chrtime}_{\mathcal{M}}(n) = \max\{chrtime_{\mathcal{M}}(\mathbb{G}) \mid \mathbb{G} \in \mathcal{VG} \wedge inputsize(\mathbb{G}) = n\}$$

*where inputsize is a function which returns the size of a goal*[1]

Note that the definition of $chrtime_{\mathcal{M}}$ does not correspond to what is obtainable in real CHR implementations, because finding an applicable rule with $k$ heads in a store of size $n$ may take up to $\mathcal{O}(n^k)$ time.

**Definition 11 (State size function).**

$$\textsc{size} : \Sigma^{\text{CHR}} \to \mathbb{N} : \langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \mapsto size(\mathbb{G}) + size(\mathbb{S}) + size(\mathbb{B}) + size(\mathbb{T})$$

*where for sets $X$, $size(X) = \sum_{x \in X} |x|$ and the size $|x|$ is the usual term size.*

**Definition 12.** *Given a CHR machine $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{VG})$, the* space function *$chrspace_{\mathcal{M}}$ returns the worst-case execution state size, given an initial goal:*

$$chrspace_{\mathcal{M}}(\mathbb{G}) = \max\{\textsc{size}(\sigma) \mid \sigma \in\in \Delta^{\mathcal{H}}_{\omega_t}|_{\mathbb{G}}\}.$$

**Definition 13.** *Given a CHR machine $\mathcal{M} = (\mathcal{H}, \mathcal{P}, \mathcal{VG})$, the (worst-case) space complexity function* $\textsc{chrspace}_{\mathcal{M}}$ *is defined as follows:*

$$\textsc{chrspace}_{\mathcal{M}}(n) = \max\{chrspace_{\mathcal{M}}(\mathbb{G}) \mid \mathbb{G} \in \mathcal{VG} \wedge inputsize(\mathbb{G}) = n\}$$

Note that if the size of individual constraints is bounded, the size of the constraint store is asymptotically dominated by the number of **Introduce** steps, and the size of the built-in store is dominated by the number of **Solve** steps of the $\omega_t$ operational semantics.

[1] The function *inputsize* can be problem-specific: for instance, depending on the problem at hand, if the input is an integer number $x$ (wrapped in some constraint), the input size function could be defined as the number of bits needed to represent $x$, or just as the number $x$ itself — this choice of course dramatically influences the resulting complexity. From now on, we will assume that the usual term size is used.

### 2.5 Complexity of CHR Machines

In [2] we have demonstrated that there is a deterministic abstract CHR machine that can simulate RAM machines (and hence also Turing machines) without any overhead. This means that "everything can be done efficiently in CHR".

**Theorem 1 (Theorem 4.14 in [2]).** *For any sufficiently strong host language $\mathcal{H}$, a CHR($\mathcal{H}$) machine $\mathcal{M}_{\mathrm{RAM}}$ exists which can simulate, in $\mathcal{O}(T+P+S)$ time and $\mathcal{O}(S+P)$ space, a $T$-time, $S$-space standard RAM machine with a program of $P$ lines.*

We have also demonstrated that everything can be done efficiently in CHR *in practice*, in the sense that the CHR program can be executed in existing implemented CHR systems with the right time and space complexity:

**Theorem 2 (Corollary 5.10 in [2]).** *For every (RAM machine) algorithm which uses at least as much time as it uses space, a CHR program exists which can be executed in the K.U.Leuven CHR system, with time and space complexity within a constant from the original complexities.*

## 3 Strategy Classes

The $\omega_t$ operational semantics is very nondeterministic, in the sense that for most programs, the number of possible derivations is very large. This is of course the reason why the confluence property is crucial for program correctness.

However, all CHR implementations somehow restrict the nondeterminism of the $\omega_t$ semantics. While still guaranteeing rule application until exhaustion, they usually impose some order in which rules are tried. In effect, they instantiate the $\omega_t$ semantics; the best-known such instantiation is the refined operational semantics $\omega_r$ [7].

In this section we examine instantiations of the $\omega_t$ semantics in a general formal framework. We hope that this will lead to more insight and intuition, and this framework may also be used to study the effects of differences in implementations.

### 3.1 Execution Strategies

**Definition 14.** *An* execution strategy *fixes the output for every initial state. Formally, $\xi$ is an execution strategy for a program $\mathcal{P}$ if $\xi \subseteq \Delta^{\mathcal{H}}_{\omega_t}(\mathcal{P})$ and $\rightsquigarrow_\xi$ is a total function over $\Sigma^{init}$, i.e.*

$$\forall x, y, z \in \Sigma^{\mathrm{CHR}} : x \rightsquigarrow_\xi y \wedge x \rightsquigarrow_\xi z \Rightarrow y = z \tag{1}$$

$$\forall \sigma \in \Sigma^{init} : \exists \sigma' \in \Sigma^{\mathrm{CHR}} \cup \{\infty\} : \sigma \rightsquigarrow_\xi \sigma' \tag{2}$$

The set of all execution strategies for a program $\mathcal{P}$ is denoted by $\Omega^{\mathcal{H}}_t(\mathcal{P})$. Clearly not every execution strategy can be implemented. For instance, with the right execution strategy, the following program solves the halting problem:

```
check_halts(TM) <=> true.
check_halts(TM) <=> fail.
```

Clearly we need a more realistic notion of execution strategies.

**Definition 15.** *A* computable *execution strategy $\xi$ is an execution strategy for which the following objects exist: a set of* concrete states $\mathcal{C}\Sigma$*, a computable*[2] abstraction *function* $\alpha : \mathcal{C}\Sigma \to \Sigma^{\mathrm{CHR}}$*, a computable* initialization *function* $\beta : \Sigma^{init} \to \mathcal{C}\Sigma$ *such that* $\forall \sigma \in \Sigma^{init} \alpha(\beta(\sigma)) = \sigma$*, and a computable partial* concrete transition *function* $\mathcal{C}t : \mathcal{C}\Sigma \to \mathcal{C}\Sigma$*, such that,* $\forall x \in \Sigma^{init}$*:*

$$x \rightsquigarrow_\xi y \iff \exists n \in \mathbb{N} : \alpha(\mathcal{C}t^n(\beta(x))) = y \text{ and } \mathcal{C}t^{n+1}(\beta(x)) \text{ is undefined}$$

### 3.2 Strategy Classes

An execution strategy completely determines the result of any query. In general, although the result set may be smaller than that of all $\omega_t$ derivations, most CHR systems are still not completely deterministic. For instance, the refined operational semantics does not fix the order of constraint reactivation and partner constraint matching, which could lead to different results for the same query. However, a specific version of a CHR system, possibly with specific compiler optimizations turned on or off, should be completely deterministic, so it has only one execution strategy for every CHR program.

**Definition 16.** *A* strategy class $\Omega(\mathcal{P}) \subseteq \Omega_t^{\mathcal{H}}(\mathcal{P})$ *is a set of execution strategies for* $\mathcal{P}$*.*

As an example, there is a strategy class corresponding to the K.U.Leuven CHR system (which may contain more than one execution strategy depending on the version and the settings of the optimization flags), and it is a subset of the strategy class corresponding to the refined operational semantics. Many CHR implementations are (possibly different) instantiations of the refined semantics, in the sense that their strategy class is a subset of $\Omega_r^{\mathcal{H}}$.

Note that strategy classes are subsets of the power set of $\Delta_{\omega_t}^{\mathcal{H}}$. We will sometimes drop the argument $(\mathcal{P})$ to avoid heavy notation.

**Definition 17.** *A* computable *strategy class is a strategy class which contains at least one computable execution strategy.*

Clearly, the K.U.Leuven CHR strategy class is computable, which implies that the refined semantics is also a computable strategy class, and so is the abstract semantics.

[2] A function is computable if there is a Turing machine that computes it.

**The refined operational semantics $\omega_r$.** We define $\Sigma_r^{\text{CHR}}$ to be the set of execution states of the refined semantics [7], and $\Omega_r^{\mathcal{H}}(\mathcal{P})$ as the strategy class corresponding to $\omega_r$ derivations of a program $\mathcal{P}$.

Not all execution strategies in $\Omega_r^{\mathcal{H}}(\mathcal{P})$ are computable. As an example, consider the following program:

```
check_halts(TM) <=> answer(true), answer(false), choose.
choose, answer(X) <=> do(X).
do(false) <=> fail.
```

With an appropriate $\omega_r$ execution strategy, this program solves the halting problem. Since the second rule is applied with `choose/0` as the active constraint, the refined semantics does not fix the choice of partner constraint. The execution strategy that always picks the correct answer is obviously not computable.

To show that $\Omega_r^{\mathcal{H}}(\mathcal{P})$ is a computable strategy class, it suffices to identify one computable execution strategy in $\Omega_r^{\mathcal{H}}(\mathcal{P})$. Following [7], we can define an abstraction function $\alpha$ which maps $\Sigma_r^{\text{CHR}}$ to $\Sigma^{\text{CHR}}$:

$$\alpha(\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n) = \langle no\_id(\mathbb{A}), \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$$

where $no\_id(\mathbb{A}) = \{c \mid c \in \mathbb{A} \text{ is not of the form } c\#i \text{ or } c\#i : j\}$. The set of concrete states is $\Sigma_r^{\text{CHR}}$ and the initialization function is the identity function.

Now we still have not given a computable execution strategy, which needs to have a transition relation that is a computable function, that is, every state has a unique next state, and the latter can be computed from the former. The transition relation of $\omega_r$ is not a function, so we consider a subset of it which is a function, for instance, the function which maps every execution state in $\sigma \in \Sigma_r^{\text{CHR}}$ to the lexicographically first element in $\{\sigma' \mid \sigma \rightarrowtail^{\omega_r} \sigma'\}$. Since the set of all next $\omega_r$ states is computable, and the lexicographically first element of a set is computable, this function is computable.

**The priority semantics $\omega_p$.** We define $\Omega_p^{\mathcal{H}}(\mathcal{P})$ as the strategy class corresponding to derivations in the priority semantics $\omega_p$ [9]. We denote an assignments of priorities to rules with $\bar{p}$, and we write $\Omega_p^{\mathcal{H}}(\mathcal{P}, \bar{p})$ for the subset of $\Omega_p^{\mathcal{H}}(\mathcal{P})$ which corresponds to $\omega_p$ derivations with the priority assignments $\bar{p}$.

Again, $\Omega_p^{\mathcal{H}}(\mathcal{P})$ contains non-computable execution strategies as well as computable ones.

### 3.3 Generalized Confluence

We now generalize the definition of confluence to arbitrary strategy classes:

**Definition 18 ($\Omega$-confluence).** *A CHR program $\mathcal{P}$ is $\Omega(\mathcal{P})$-confluent if, for every initial state $\langle \mathbb{G}, \emptyset, true, \emptyset \rangle_1 = \sigma \in \Sigma^{init}$ and arbitrary execution strategies $\xi_1, \xi_2 \in \Omega(\mathcal{P})$, the following holds:*

$$\left.\begin{array}{c} \sigma \rightsquigarrow_{\xi_1} \langle \mathbb{G}_1, \mathbb{S}_1, \mathbb{B}_1, \mathbb{T}_1 \rangle_{n_1} \\ \wedge \\ \sigma \rightsquigarrow_{\xi_2} \langle \mathbb{G}_2, \mathbb{S}_2, \mathbb{B}_2, \mathbb{T}_2 \rangle_{n_2} \end{array}\right\} \Rightarrow \mathcal{D}_{\mathcal{H}} \models \bar{\exists}_{\mathbb{G}}(\mathbb{S}_1 \wedge \mathbb{B}_1) \leftrightarrow \bar{\exists}_{\mathbb{G}}(\mathbb{S}_2 \wedge \mathbb{B}_2)$$

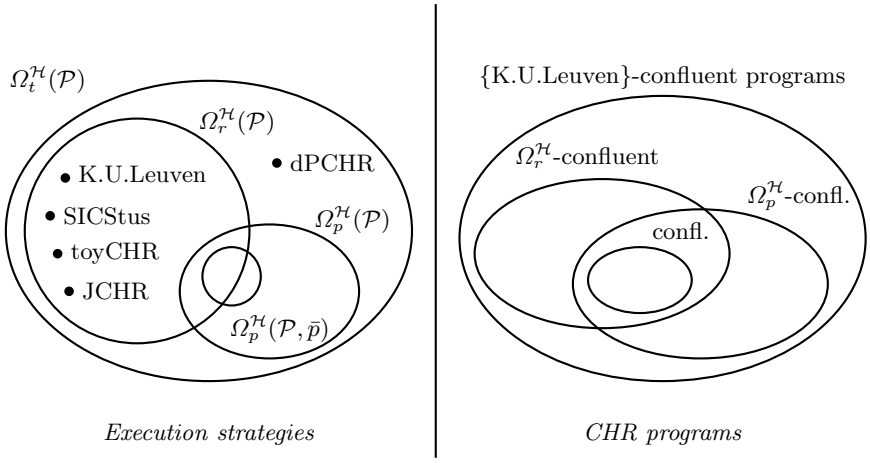



**Fig. 2.** Execution strategies and CHR programs.

Note that $\Omega_t^{\mathcal{H}}(\mathcal{P})$-confluence is the same as the usual confluence (per definition), while $\Omega_r^{\mathcal{H}}(\mathcal{P})$-confluence corresponds to "confluent in the refined semantics" (see also [10], chapter 6). If the strategy class $\Omega$ is a singleton, every CHR program $\mathcal{P}$ is trivially $\Omega$-confluent.

Figure 2 shows some strategy classes and the corresponding sets of $\Omega$-confluent programs. The execution strategy "dPCHR" is that of an implementation of probabilistic CHR [11], in which all rules get the same probability and the rules are picked using a deterministic pseudo-random number generator initialized with some fixed seed.

We have the following duality property that follows directly from the definitions: for all strategy classes $\Omega_1$ and $\Omega_2$: if $\Omega_1 \subseteq \Omega_2$, then every $\Omega_2$-confluent program is also $\Omega_1$-confluent. In other words, if $\Omega_2$ is more general than $\Omega_1$ (i.e. it allows more derivations), then $\Omega_2$-confluence is stronger than $\Omega_1$-confluence.

## 4 General CHR Machines

We generalize the deterministic abstract CHR machines of Section 2.2 as follows:

**Definition 19.** *A* CHR machine *is a tuple* $\mathcal{M} = (\mathcal{H}, \Omega, \mathcal{P}, \mathcal{VG})$ *where the host-language* $\mathcal{H}$ *defines a built-in constraint theory* $\mathcal{D}_{\mathcal{H}}$*,* $\mathcal{P}$ *is a CHR program,* $\mathcal{VG} \subseteq \mathcal{G}_{\mathcal{P}}^{\mathcal{H}}$ *is a set of* valid goals*, and* $\Omega \subseteq \Omega_t^{\mathcal{H}}(\mathcal{P})$ *is a strategy class. The machine takes an input query* $\mathbb{G} \in \mathcal{VG}$*, picks any execution strategy* $\xi \in \Omega$*, and executes a derivation* $d \in \xi|_{\mathbb{G}}$*.*

Note that we no longer require the program to be $\Delta_{\omega_t}^{\mathcal{H}}$-deterministic for valid input, and we allow it to use any strategy class.

**Terminology.** A *CHR($\mathcal{H}$) machine* is a CHR machine of the form $(\mathcal{H}, \_, \_, \_)$. An *$\Omega$-CHR machine* is a CHR machine of the form $(\_, \Omega, \_, \_)$. *Abstract* CHR machines are $\Omega_t^{\mathcal{H}}$-CHR machines, and *refined* CHR machines are $\Omega_r^{\mathcal{H}}$-CHR machines. A *feasible* CHR machine is one with a computable strategy class. A *confluent* CHR machine $(\_, \Omega, \mathcal{P}, \_)$ has a program $\mathcal{P}$ which is $\Omega$-confluent. A *deterministic* CHR machine $(\_, \Omega, \mathcal{P}, \_)$ has a program $\mathcal{P}$ which is $\Delta_\Omega$-deterministic, where $\Delta_\Omega = \bigcup \Omega(\mathcal{P})$ is the set of all possible derivations.

We generalize the definitions of the time and space functions in the straightforward way:

**Definition 20.** *Given an $\Omega$-CHR machine $\mathcal{M}$, the* time function *$chrtime_{\mathcal{M}}$ returns the worst-case derivation length, given an initial goal $\mathbb{G} \in \mathcal{VG}$:*

$$chrtime_{\mathcal{M}}(\mathbb{G}) = \max\{\#d \mid \exists \xi \in \Omega : d \in \xi|_{\mathbb{G}}\}$$

**Definition 21.** *Given an $\Omega$-CHR machine $\mathcal{M}$, the* space function *$chrspace_{\mathcal{M}}$ returns the worst-case execution state size, given an initial goal:*

$$chrspace_{\mathcal{M}}(\mathbb{G}) = \max\{\textsc{size}(\sigma) \mid \exists \xi \in \Omega : \sigma \in\in \xi|_{\mathbb{G}}\}$$

It is not clear what the added power is of generalized CHR machines compared to CHR machines that follow the abstract operational semantics. For well-known strategy classes, like $\Omega_r^{\mathcal{H}}$ or $\Omega_p^{\mathcal{H}}$, we can still only decide the languages in P in polynomial time. However, it seems that more instantiated strategy classes add some power. For example, in the refined semantics, we can (non-monotonically) check for absence of constraints, and in the priority semantics, we can easily sort in linear time. We can imagine more exotic strategy classes, that could still be computable while implicitly requiring more than a polynomial amount of work to compute the next transition. For such strategy classes, the corresponding generalized CHR machine could of course be much more powerful.

## 5 Non-deterministic CHR Machines

We define *non-deterministic* CHR machines similarly to the way non-deterministic Turing machines are defined.

**Definition 22.** *A* non-deterministic CHR machine *(NCHR machine) is a tuple $\mathcal{M} = (\mathcal{H}, \Omega, \mathcal{P}, \mathcal{VG})$, where $\mathcal{H}$, $\Omega$, $\mathcal{P}$, and $\mathcal{VG}$ are defined as before. The machine takes an input query $\mathbb{G} \in \mathcal{VG}$ and considers all execution strategies $\xi \in \Omega$. If there are strategies that result in a successful derivation $d \in \xi|_{\mathbb{G}}$, any of those is returned. Otherwise, any failure derivation is returned. If all derivations are infinite, any infinite derivation is returned.*

As an example, consider the following CHR program $\mathcal{P}_{\text{3SAT}}$:

```
clause(A,_,_) <=> true(A).
clause(_,B,_) <=> true(B).
clause(_,_,C) <=> true(C).
true(X), true(not(X)) <=> fail.
```

The NCHR machine $(\emptyset, \Omega_t^{\mathcal{H}}, \mathcal{P}_{\text{3SAT}}, \text{3SATCLAUSES})$ decides 3SAT clauses in linear time. A 3SAT clause of the form $(x_1 \vee x_2 \vee x_3) \wedge (y_1 \vee y_2 \vee y_3) \wedge \ldots$ is encoded as a query `clause(x1,x2,x3), clause(y1,y2,y3), ...`, where negative literals are encoded by wrapping them in `not/1`. The valid goals 3SATCLAUSES (i.e. admissible input clauses, but not necessarily satisfiable) are all goals of this form.

In every derivation, all `clause/3` constraints are simplified, so one of the literals has been made true. If the instance of 3SAT has a solution, there is a way to do this without producing a conflicting truth assignment, so there is a successful derivation. If there is no solution, all derivations will fail because every assignment causes a conflict, which fires the fourth rule.

The NCHR machine $(\emptyset, \Omega_r^{\mathcal{H}}, \mathcal{P}_{\text{3SAT}}, \text{3SATCLAUSES})$ — the same as above but with its strategy class restricted to the refined semantics — is no longer correct. Because execution strategies are limited to those of the $\omega_r$ semantics, every `clause/3` constraint will be simplified by the first rule. As a result, the only truth assignment that is tried is to make every first literal of all clauses true.

Non-determinism in rule choice is exploited in the program $\mathcal{P}_{\text{3SAT}}$. However, we can easily transfer the non-determinism to the choice of matching partner constraints for an active constraint. For example, the following program, when executed on a refined NCHR machine, decides 3SAT clauses in linear time:

```
clause(A,B,C) <=> d(X,A), d(X,B), d(X,C), c(X).
c(X), d(X,A) <=> true(A).
true(X), true(not(X)) <=> fail.
```

However, as a general rule, the smaller the strategy class, the harder it is to write a correct NCHR program: when there are less sources of non-determinism, the corresponding NCHR machine becomes less powerful. When the strategy class is a singleton, there is of course no difference between a regular CHR machine and an NCHR machine.

For regular CHR machines, the reverse rule of thumb holds: the larger the strategy class, the harder it is to write a correct CHR program — more non-determinism only means more wrong choices. If we denote the class of decision problems that can be solved by a deterministic $\Omega$-CHR machine in polynomial time with $P_\Omega$, and the class of decision problems that can be solved by a polynomial-time non-deterministic $\Omega$-CHR machine with $NP_\Omega$, then we have the following inclusions:

$$P_{\Omega_t^{\mathcal{H}}} \subseteq P_{\Omega_r^{\mathcal{H}}} \subseteq P_{\{\text{K.U.Leuven}\}} = NP_{\{\text{K.U.Leuven}\}} \subseteq NP_{\Omega_r^{\mathcal{H}}} \subseteq NP_{\Omega_t^{\mathcal{H}}}$$

Most of these inclusions collapse to equalities: since the RAM simulator program of [2] is ($\Omega_t^{\mathcal{H}}$-) confluent, we have $P_{\Omega_t^{\mathcal{H}}} = P_{\{\text{K.U.Leuven}\}}$. In this sense, the strategy class does not seem to affect the computational power of the CHR machine. Still, it is our experience that it is easier to write programs for a more

instantiated operational semantics. In the case of the self-modifying CHR machines of the next section, it does seem to be the case that instantiating the operational semantics really adds power to the machine.

The class of languages that are decided in polynomial time by a non-deterministic CHR machine (refined or abstract) coincides with NP. We only prove the inclusion $NP_{\Omega_r^{\mathcal{H}}} \subseteq NP$:

**Theorem 3.** *The non-deterministic refined CHR-only machine can simulate a non-deterministic Turing machine with the same complexity.*

*Proof.* The simulator program TMSIM (Fig. 1 page 146) also works if `delta/5` is not a function but defines more than one transition for a given state and symbol. The non-determinism in choosing the partner constraint for `head/1` (the computation-driving active constraint) ensures that all Turing machine computation paths are simulated.

In the above, the word "refined" can also be replaced by "abstract".

## 6 Self-modifying CHR Machines

The RASP machine (random access stored program) [12] is essentially a RAM machine which can access and change its own program instructions — much like real computers which follow the von Neumann architecture: instructions and data are stored in the same memory space, hence the term *stored program*. In terms of computational power and complexity, the RASP machine is just as powerful as a regular RAM machine; the reason is that you can write a RASP simulator on a RAM machine, which takes only a constant factor more time and space.

In this section, we examine CHR machines with a stored program, or CHRSP machines. Since the CHR program is now stored in the CHR store, it can be accessed and modified like any other CHR constraints.

### 6.1 Definition

We use a syntax that somewhat resembles that of [13], where it was proposed in the context of source-to-source transformation. A CHR program is encoded using "reserved keyword" constraints `rule/1`, `khead/2`, `rhead/2`, `guard/2`, and `body/2`. For example, the rules

```
foo @ bar ==> baz.
qux @ blarg \ wibble <=> flob | wobble.
```

would be encoded as

```
rule(foo), khead(foo,bar), guard(foo,true), body(foo,baz),
rule(qux), khead(qux,blarg), rhead(qux,wibble),
    guard(qux,flob), body(qux,wobble)
```

**3. Apply.** $\langle \mathbb{G}, H_1 \cup H_2 \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle C' \uplus \mathbb{G}, H_1 \cup \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$
where $chr(\mathbb{S})$ contains the following constraints, which encode a rule: `rule(`$r$`),` `khead(`$r$`,`$h_1$`), ..., khead(`$r$`,`$h_k$`),` `rhead(`$r$`,`$h_{k+1}$`), ..., rhead(`$r$`,`$h_l$`),` `guard(`$r$`,`$g$`),` `body(`$r$`,`$C$`)`, and neither $\mathbb{G}$ nor $\mathbb{S}$ contain any other rule-encoding constraints with $r$ as a first argument. Now let $g'$, $C'$, $H'_1$, and $H'_2$ be consistently renamed apart versions of $g$, $C$, $(h_1 \wedge \cdots \wedge h_k)$, and $(h_{k+1} \wedge \cdots \wedge h_l)$, respectively. This encoding corresponds to a rule of the form $r \mathbin{@} H'_1 \setminus H'_2 \iff g' \mid C'$. As usual, $\theta$ is a matching substitution such that $chr(H_1) = \theta(H'_1)$ and $chr(H_2) = \theta(H'_2)$ and $h = (r, id(H_1), id(H_2)) \notin T$ and $\mathcal{D}_{\mathcal{H}} \models (\exists \mathbb{B}) \wedge (\mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}}(\theta \wedge g'))$.

**Fig. 3.** The new **Apply** transition in the $\omega_t^{sp}$ semantics for CHRSP machines

Now, we modify the **Apply** transition of $\omega_t$ to refer to the stored program as in Fig. 3. Note that because the program is now in the constraint store, we no longer need to (implicitly) parametrize the semantics with a CHR program; the program is now part of the query or, equivalently, corresponds to the initial store. In order to avoid premature application of rules, i.e. firing a rule which is still being constructed, we require that a rule can only fire if there are none of its components in the goal, waiting to be introduced into the store.

A CHRSP machine is defined just like a regular CHR machine, except that the operational semantics is altered in the way described above. Also, if a program is given for CHRSP machines, it should be considered to be an abbreviation for the encoded form, which is implicitly appended to all valid goals. As before, every $\omega_t^{sp}$ transition takes constant time; the machine rejects the input if the final state is a failure state, otherwise it accepts or does not terminate.

### 6.2 Complexity of CHRSP Machines

Unlike RASP machines, which can be efficiently simulated on RAM machines, CHRSP machines cannot be simulated on regular CHR machines with only constant overhead. The reason is that finding $k$ partner constraints in a store of size $n$ can take $\mathcal{O}(n^k)$ time. For a fixed program, $k$ is bounded, but on a CHRSP machine, rules with an arbitrary number of heads may be created.

In fact, self-modifying CHR programs can decide co-NP-complete languages in only linear time. Consider the problem of Hamiltonian paths in directed graphs. Deciding whether a graph has a Hamiltonian path is NP-complete; the language of consisting of all graphs that do not have a Hamiltonian path is therefor co-NP-complete. Now consider the following self-modifying CHR program:

```
size(N) <=> rule(find_path), size(N,A).
size(N,A) <=> N>1 |
    khead(find_path,node(A)),
    khead(find_path,edge(A,B)),
    size(N-1,B).
size(1,A) <=>
```

```
khead(find_path,node(A)),
body(find_path,fail).
```

As input query, we encode a graph in the following way: `edge/2` constraints for the edges; `node/1` constraints for the $n$ nodes; one `size(`$n$`)` constraint to indicate the number of nodes. The program will create a rule of the form

```
find_path @ node(A1), node(A2), ..., node(An),
      edge(A1,A2),edge(A2,A3), ..., edge(An-1,An) ==> fail.
```

If the graph has a Hamiltonian path, this rule fires and the CHRSP machine rejects the input. Otherwise, the machine accepts the input. Either way, the machine halts after $\mathcal{O}(n)$ steps.

If a regular CHR machine exists that can simulate CHRSP machines with only polynomial overhead, then co-NP $\subseteq$ P, and thus P = NP. So if $P \neq NP$, CHRSP machines are strictly more powerful than regular CHR machines.

## 7 Summary and Conclusion

We have defined three different generalizations of the CHR machine of [2]: CHR machines with restricted strategy classes, non-deterministic CHR machines, and stored-program (self-modifying) CHR machines. These generalizations are orthogonal, so they can be combined. Indeed, one could very well consider, for instance, a refined self-modifying CHR machine. We have investigated the complexity properties of these generalized CHR machines.

### 7.1 Complexity Summary

As shown in [2], a regular CHR machine can do in polynomial time what a Turing machine (or a RAM machine) can do in polynomial time:

$$P_{\Omega_t} = P$$

In Section 5 we showed a similar result for non-deterministic CHR machines:

$$NP_{\Omega_t} = NP$$

However, although $P_{\mathrm{RASP}} = P$, self-modifying CHR machines are more powerful than regular ones, although exact bounds are still an open problem:

$$coNP \subseteq P_{\Omega_t^{sp}} \subseteq PSPACE$$

Restricting the strategy class to an instantiation of $\Omega_t$ (or $\Omega_t^{sp}$) can make the CHR machine stronger: a self-modifying *refined* CHR machine can also solve $NP$-complete problems in linear time by checking for absence of a solution to the corresponding $coNP$ problem. Checking for absence is not known to be possible in the abstract semantics. So $P_{\Omega_t^{sp}} \subseteq P_{\Omega_r^{sp}}$ (and we conjecture the inclusion to be strict), and also $coNP \cup NP \subseteq P_{\Omega_r^{sp}}$.

### 7.2 Future Work

Determining the exact complexity classes corresponding to CHRSP machines is still an open problem. It is also not clear to what extent the choice of strategy class influences the computational power, even for regular generalized CHR machines, let alone in the case of non-deterministic and/or stored-program CHR machines.

## References


1. Sneyers, J., Schrijvers, T., Demoen, B.: The computational power and complexity of Constraint Handling Rules. In Schrijvers, T., Frühwirth, T., eds.: CHR '05. K.U.Leuven, Dept. Comp. Sc., Technical report CW 421, Sitges, Spain (2005) 3–17
2. Sneyers, J., Schrijvers, T., Demoen, B.: The computational power and complexity of Constraint Handling Rules. To appear in ACM TOPLAS (2008)
3. Frühwirth, T.: Theory and practice of Constraint Handling Rules. J. Logic Programming, Special Issue on Constraint Logic Programming **37**(1–3) (1998) 95–138
4. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2008) To appear.
5. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules — a survey of CHR research between 1998 and 2007. Submitted to Theory and Practice of Logic Programming (2008)
6. Giusto, C.D., Gabbrielli, M., Meo, M.C.: Expressiveness of multiple heads in CHR. Submitted to PPDP'08 (2008)
7. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In Demoen, B., Lifschitz, V., eds.: ICLP '04. Volume 3132 of LNCS., Saint-Malo, France, Springer (September 2004) 90–104
8. Duck, G.J., Stuckey, P.J., Sulzmann, M.: Observable confluence for Constraint Handling Rules. In Dahl, V., Niemelä, I., eds.: ICLP '07. Volume 4670 of LNCS., Porto, Portugal, Springer (September 2007) 224–239
9. De Koninck, L., Schrijvers, T., Demoen, B.: User-definable rule priorities for CHR. In Leuschel, M., Podelski, A., eds.: PPDP '07, Wrocław, Poland, ACM Press (July 2007) 25–36
10. Duck, G.J.: Compilation of Constraint Handling Rules. PhD thesis, University of Melbourne, Victoria, Australia (December 2005)
11. Frühwirth, T., Pierro, A.D., Wiklicky, H.: Probabilistic Constraint Handling Rules. In Comini, M., Falaschi, M., eds.: WFLP '02: Proc. 11th Intl. Workshop on Functional and (Constraint) Logic Programming, Selected Papers. Volume 76 of ENTCS., Grado, Italy, Elsevier (June 2002)
12. Hartmanis, J.: Computational complexity of random access stored program machines. Theory of Computing Systems **5**(3) (September 1971) 232–245
13. Frühwirth, T., Holzbaur, C.: Source-to-source transformation for a class of expressive rules. In Buccafurri, F., ed.: AGP '03: Joint Conf. Declarative Programming APPIA-GULP-PRODE, Reggio Calabria, Italy (September 2003) 386–397

# Prioritized Abduction with CHR

Henning Christiansen

Research group PLIS: Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: henning@ruc.dk

**Abstract.** A class of Prioritized Abductive Logic Programs (PrioALPs) is introduced and an implementation is developed in CHR for solving abductive problems, providing minimal explanations with best-first search. Priorities may represent probabilities or a cost function to be optimized. Compared with other weighted and probabilistic versions of abductive logic programming, the approach is characterized by higher generality and a flexible and adaptable architecture which incorporates integrity constraints and interaction with external constraint solvers.
A PrioALP is translated in a systematic way into a CHR program which serves as a query interpreter, and the resulting CHR code describes in a highly concise way, the strategies applied in the search for explanations.

## 1 Introduction

Metaprogramming for implementation of advanced logic programming paradigms is one established class of applications of CHR [1] that exceeds the original goal of writing traditional constraint solvers in a declarative way; we may refer to earlier work such as [2–6]. In particular, abduction in logic programs has been implemented in an efficient way using a combination of Prolog and CHR, where an abductive logic program is executed as a Prolog program with CHR taking care of abducibles and integrity constraints [5, 7]; however, these approaches do not at present provide interesting use of negation. Such implementations of abduction inherit Prolog's depth-first search strategy.

Using instead CHR's constraint store to hold a pool of pending processes, it is possible to implement advanced breadth-first like strategies; [8] considers a paradigm where the control jumps back and forth between different branches depending on the content of a global knowledge base common to all branches (Global Abduction [9, 10]); [11] implements a probabilistic version of CHR in which the choice of rule to be applied is taken in a probabilistic way.

In the present paper, we study the use of the process-pool principle for best-first search in abduction, referring to an arbitrary, user-defined priority function. Probabilistic abduction is encoded using a probability distribution as priority function. More precisely, we consider a class of Prioritized Abductive Logic Programs with integrity constraints. and where abducibles and explanations (sets of abducibles) are assigned priorities. Best-first means that the execution follows

the branch whose partial explanation has highest priority, which means that the control may jump back to an earlier delayed branch, if that looks currently best. It is also demonstrated how these programs can refer to existing constraint solvers for a further pruning of the search space.

**Related work.** We have referred to earlier work on abduction in CHR above. Different paradigms for prioritized, weighted and probabilistic abduction have been studied in different contexts, e.g., [12–14]. The mentioned approaches do not consider general integrity constraints and integration with constraint solvers. Poole [13] defines a class of weighted abductive logic programs different from ours and explains a best-first implementation as a metainterpreter in Prolog; among abductive systems without priorities, [15–17] describe systems which interact with specific constraint solvers, and [18] indicates an implementation in CHR of a system which can handle some cases of negation outside the scope of the present and other CHR based approaches referenced. We shall refrain from giving detailed references to the tradition of Bayesian networks, which can be seen as a highly restricted case of probabilistic abduction (from a semantic point of view), but for which efficient methods exist that can handle very large data sets. For an overview of abductive logic programming in general, we refer to [19, 20]. An advanced, rule-based system for defining preferences among abducibles is proposed by [21], and it is not clear at present whether this can be expressed in the framework presented below.

**Overview.** Section 2 defines prioritized abductive programs, and section 3 describes the architecture of their best-first query interpreters in CHR and Prolog. Probabilistic abduction, as a special case of prioritized abduction, is explained in section 4; section 5 exemplifies how external constraint solvers can be integrated; finally, section 6 indicate further optimizations inspired by Dijkstra's shortest path algorithm and by simplifying integrity constraints.

## 2 Definition of Prioritized Abduction

**Definition 1.** *A* prioritized abductive logic program (PrioALP) *is given by*

- *a set of predicate symbols, each with a fixed arity, distinguished into four disjoint classes,* abducibles*,* program defined*,* external *and* $\bot$*,*
- *a priority function $F$ which maps sets of abducible atoms into a totally ordered set $\langle \mathcal{P}, < \rangle$ (with $\leq$ defined in the usual way) such that $F(A) \geq F(A \cup B)$ and $F(A) \geq F(A\sigma)$ for any substitution $\sigma$,*
- *a set of clauses of the form* $A_0$`:-`$A_1$`,`$\ldots$`,`$A_n$*, of which the following kinds are possible,*
  - ordinary clauses *where $A_0$ is an atom of a program defined predicate and none of $A_1, \ldots, A_n$, $n \geq 0$, are $\bot$,*
  - integrity constraints *in which $A_0 = \bot$ and $A_1, \ldots, A_n$, $n \geq 1$, are abducible atoms.*

*As usual, an arbitrary and infinite collection of function symbols, including constants, are assumed, and atoms are defined in the standard way.* □

The relationship $\models$ refers to the usual, completion-based semantics for logic programs; for external predicates, we assume a semantics independent of the actual program, and without specifying further, an a priori defined truth value for $\models e$ is given for any ground external atom $e$. In practice, external predicates can be Prolog built-ins or defined by additional Prolog clauses, or constraints given either by a Prolog library or by additional CHR rules. We need to require that any call to an external predicate always succeeds at most once; until section 5, constraints are excluded for simplicity.

When $\Pi$ is a PrioALP and we write $\Pi \models \cdots$ or $\Pi \cup \cdots \models \cdots$, we use $\Pi$ to refer to the completion of all clauses of $\Pi$ including the integrity constraints and (for brevity) a theory defining a meaning for all external predicates. When no ambiguity arises, a clause is usually an ordinary clause, and integrity constraints will be referred to as such. The notation $[\![F_1, \ldots, F_n]\!]$, $F_i$ being formulas, is taken as a shorthand for $\exists(F_1 \wedge \cdots \wedge F_n) \wedge \neg\bot$. Notice the following trivial properties,

$$[\![A \wedge B]\!] \equiv [\![A]\!] \wedge [\![B]\!] \tag{1}$$

$$[\![A \vee B]\!] \equiv [\![A]\!] \vee [\![B]\!] \tag{2}$$

which mean that any standard distributive law that does not involve negation can be used for formulas within $[\![-]\!]$.

**Definition 2.** *A* query *or* goal *is a conjunction of non-$\bot$ atoms; a set of ground abducible atoms is called a* state*; a set of (not necessarily ground) abducible atoms is called a* state term*. In the context of a PrioALP $\Pi$, we say that state or state term $S$ is* inconsistent *whenever $\Pi \cup \forall S \models \bot$ and otherwise* consistent*. For two state terms $S_1, S_2$, we say that $S_1$* subsumes *$S_2$ and that $S_1$ is* more general than *$S_2$, whenever*

$$\models \exists S_1 \leftarrow \exists S_2. \tag{3}$$

*Whenever $S_1$ subsumes $S_2$ and vice-versa, we say that they are* equivalent*.*

*Given a PrioALP $\Pi$ and a query $Q$, an* explanation *for $Q$ is a state term $E$ such that*

$$\Pi \cup \exists E \models [\![Q]\!] \tag{4}$$

*An explanation $E$ for $Q$ is* minimal *if it is not a subsumed by any other explanation for $Q$.* □

Notice that an explanation $E$ with smallest priority, i.e., there is no other $E'$ with $F(E') < F(E)$, is also minimal. We say that an explanation with higher (highest) priority number is *better (best).* Whenever $E$ subsumes $E'$, we have that $F(E') > F(E)$. Thus a priority function is monotonically decreasing in the sense that more commitments may lower, but never raise, the priority; probability distributions over (possibly independent) abducibles are examples of priorities.

## 3 A Generic Architecture for Best-First Implementation of Prioritized Abduction

We describe here an architecture for implementing PrioALP by a translation into CHR which, when given a query $Q$, calculates a best minimal explanation,

and, if requested by the user, more minimal explanations ordered according to their priority.

The priority function, integrity constraints, and indication of abducible and external predicates are encoded into auxiliary predicates, and program clauses are compiled into CHR rules.

### 3.1 Auxiliary Predicates

We do not need to specify a representation for explanations here but we assume there is a notion of a *reduced form* of representations; we anticipate representations as lists of abducible literals without duplicates; we assume, though, that the empty explanation is represented as `[]`. From a logical point of view, the reduced form is not interesting, but is useful for efficiency and when presenting final explanations to the user. We assume a context which includes a PrioALP so that we can refer to the notion of consistency and a priority function $P$.

`subsumes(`$E_1$`,`$E_2$`)` $\equiv E_1$ subsumes $E_2$, i.e., $\models \exists E_2 \rightarrow \exists E_1$, when $E_1, E_2$ are consistent state terms.

`entailed(`$A$`,`$E$`)` $\equiv \models \forall(E \rightarrow A)$ when $A$ is an abducible atom and $E$ a consistent state term.

`extend(`$A$`,`$E$`,`$F(E)$`,`$E'$`,`$F(E')$`)` $\equiv \models \forall(E' \leftrightarrow A \wedge E)$ when $A$ is an abducible atom and $E$, $E'$ consistent state terms in reduced form so that `entailed(`$A$`,`$E$`)` does not hold.[1]

Notice the different usages of quantifiers. For `entailed/2` and `extend/5`, the presence of common variables in the arguments are significant, and variables may be bound later in the computation, whereas `subsumes/2` concerns explanations in different branches of computation (which, in fact, will have no variables in common).

The following predicate is used whenever an explanation may be affected by unifications, which may be a consequence of applying a rule of the given PrioALP or a call to an external predicate.

`recalculate(`$E$`,`$E_1$`,`$F(E_1)$`)` $\equiv \forall(E \leftrightarrow E_1)$, $E_1$ is in reduced form, and $E$ and $E_1$ are consistent state terms.

We have introduced this predicate as it may be implemented quite efficiently; it very seldom pays off to analyze the detailed effect of a unification in order to reuse the previously calculated priority.

`abducible(`$A$`)` $\equiv A$ is an atom of an abducible predicate.

`external(`$A$`)` $\equiv A$ is an atom of an external predicates

`priority_less_than(`$P_1$`,`$P_2$`)` $\equiv P_1 < P_2$ where $P_1$ and $P_2$ are priorities and $<$ the priority ordering.

[1] The third argument of `extend`, $F(E)$, is redundant, but can be used for priority functions that allow an incremental evaluation; this is the case for probability distributions.

Finally, we need the following renaming predicates in order to create alternative variants of a query when the execution splits in different branches for alternative clauses of the given PrioALP.

`rename(`$T_1$`,`$T_2$`)` $\equiv T_2$ is a variant of $T_1$ with new variables that are not used anywhere else.

The definition of these predicates are in most cases straightforward and we will not pay much attention to this topic; versions for probabilistic priorities can be found in [22]. A few remarks, though:

- When abducibles are known to be ground at the time of call, explanations can be represented by sorted lists (by Prolog's `@<` ordering), which allows very efficient implementation of all operations on explanations.
- When nonground abducibles are considered, we cannot use sorted lists (as unifications may destroy the ordering), and we need to handle cases where abducible atoms become identical as a result of a unification; the `subsumes` and `entailed` predicates become more complicated involving skolemization.
- When built-in constraint solvers are applied as external predicates, the renaming operation should also transfer copies of constraints attached to the original term to the renamed version; see section 5.

### 3.2 Compiling Prioritized ALPs into CHR

Our overall strategy is to compile a PrioALP into a CHR program that applies the constraint store as a pool of processes, which together maintain the semantics of the initial query. These processes are transformed gradually into a final form from which minimal answers can be read out.

A process is represented as a constraint $\cdots$`_explain(`$Q$`,`$E$`,`$p$`)`, where the dots indicate that the constraint is given in different versions, for control purposes only. The meaning of such an `explain` constraint is that the query $Q$ is what remains to be proven in order to find an explanation for the initial query; $E$ is the partial explanation produced so far to get from the initial query to $Q$, and $p$ is the priority of $E$; $Q$ is here represented as a list of atomic goals. The three different variants of the `explain` constraint are the following.

- `queue_explain`: the indicated process is in the constraint store which is seen as a priority queue ordered by the priority $p$,
- `step_explain`: the process is selected to perform one step, after which the derived subprocesses are entered into the queue,
- `printed_explain`: the process has terminated. i.e., with $Q$ = `[]`, and its explanation has been presented to the user; it is needed in case the user asks for alternative explanations in decreasing order of priority.

Some rules in the query interpreter are common for all PrioALP programs, and some are specific for given program. We demonstrate below the translation for the following prototypical predicate definition.

```
p(X):- q(X,Y), r(Y).
p(X):- a(X).                                                    (5)
p(1).
```

We show the entire query interpreter for this program. A top-level predicate sets up initial parameters; for simplicity we have ignored variables in the initial query but it is straightforward to add an extra argument to the interpreter that keeps track of bindings to those variables so they can be printed out in the end.

```
explain(G):- step_explain(G,[],F([])).                          (6)
```

The following rule removes non-minimal explanations; to see that it works correctly, notice that the minimal `E1` will be produced by rules below before any extension `E2` of it as this will have lower priority.

```
printed_explain([],E1,_) \ queue_explain(_,E2,_) <=>
    subsumes(E1,E2) | true.                                     (7)
```

The currently best process in the queue is selected by the following rule, which may fire when `select_best` is called.

```
queue_explain(G,E,P)#W, select_best <=> max_prio(P) |
    step_explain(G,E,P)                                         (8)
    pragma passive(W).
```

The passive declaration does not affect the semantics of the program, but is an obvious optimization to prevent calls to `queue_explain` from considering this rule; it can only be triggered by `select_best`.

For brevity of the code, we use a straightforward and inefficient implementation of the priority queue; see [23] for a more detailed study of priority queues in CHR. The constraint `max_prio` defined by the following two rules is used, as a call in a guard below, to check if the priority of a given process is the best one.

```
max_prio(P0), queue_explain(_,_,P1)#W <=>
    priority_less_than(P0,P1) | fail
    pragma passive(W).                                          (9)
max_prio(_) <=> true.
```

While constraints in the guard of a CHR rule may lead to dubious semantics, the call to `max_prio` in (8) gives sense as it does not change the constraint store or binds variables; it is handled sensibly by most CHR implementations.

When the selected process has an empty query, it can be seen that no other process can produce an explanation with a higher priority, so it is printed out and stored as a `printed_explain` constraint for possible future use by rule (7).

```
step_explain([],E,P) <=>
    printed_explain([],E,P),
    write('Best solution: '), write(E),
    write(', Priority: '), write(P),nl,
    (user_wants_more -> select_best ; true ).
```
(10)

```
user_wants_more:-
```
*Ask user; if answer is* `y`*, succeed, otherwise fail*.

Each predicate definition, such as (5) above is translated into one CHR rule, which represents with the conjunction of the clauses; a number of different techniques and straightforward optimizations are involved in this translation. This is done by posting a new process for each clause, however, suppressing those where the unification of selected subgoal and head-of-clause fails. The latter is done by the pattern (*test* `->` *continue* `; true`), which means that a possible failure of *test* is absorbed, and the branch *continue* vanishes rather that provoking a failure in the execution of the CHR rules; this technique is also used in [8, 24]. Notice that when the arguments in the head of a clause are all variables, this is unnecessary as unification will always succeed. When a unification is performed, the auxiliary predicate `recalculate` is applied to get the new priority; in other cases the priority is inherited unchanged from the calling subgoal (`p(···)` in the example definition). Finally, notice that all variables in the current query are renamed for each clause to avoid cluttering up the different alternatives; this is suppressed for the last clause as no further use is made of these variables. Notice that `recalculate` may fail if integrity constraints are violated, in which case the relevant subprocess should vanish in the same way as when a unification fails.

```
step_explain( [p(X)|G], E, P) <=>
    rename([p(X)|G]+E,[p(Xr1)|Gr1]+E1),
    queue_explain([q(Xr1,Y),r(Y)|Gr1], E1, P),
    rename([p(X)|G]+E,[p(Xr2)|Gr2]+E2),
    queue_explain([a(Xr2)|Gr2], E2, P),
    (X=1, recalculate(E,Er,Pr) -> queue_explain(G, Er, Pr)
     ; true),
    select_best.
```
(11)

Abducible and external predicates are interpreted as follows. Notice that the priority stays the same if the abducible atom is already in the given explanation. External predicates are here treated as non-analyzable devices that may produce unifications affecting the explanation, specializing non-ground abducibles, perhaps making previously distinct ones identical. Notice the handling of a possible failure of `extend`, which may happen if integrity constraints are violated; similarly for `recalculate`.

```
step_explain( [A|G], E, P) <=> abducible(A) |
    (entailed(A,E) -> queue_explain(G, E, P)
     ;
     extend(A,E,P,E1,P1) -> queue_explain(G,E1,P1)
     ; true),
    select_best.
```
(12)

```
step_explain([X|G], E, P) <=> external(X) |
    (call(X),recalculate(E,Er,Pr) -> queue_explain(G,Er,Pr)
     ; true),
    select_best.
```
(13)

The following correctness statement, that captures soundness and completeness, can be proved by induction over CHR derivations, showing that each of the rules (6) to (13) preserve the condition (details for the probabilistic case can be found in [22]).

**Theorem 1.** *Let $\Pi$ be a PrioALP and $\Gamma$ the translation of $\Pi$ into a CHR program as described above. Consider any constraint store in a CHR derivation starting with* `explain(`$Q$`)` *for some query $Q$; the set of its $\cdots$*`_explain` *constraints can be numbered as follows,*

`printed_explain(`$Q_1$`,`$E_1$`,`$p_1$`)`
$\vdots$
`printed_explain(`$Q_k$`,`$E_k$`,`$p_k$`)`
`queue_explain(`$Q_m$`,`$E_m$`,`$p_m$`)`
$\vdots$
`queue_explain(`$Q_n$`,`$E_n$`,`$p_n$`)`

*Either $m = k+1$, or $m = k+2$ and there is an additional* `step_explain(`$Q_{k+1}$`,` $E_{k+1}$`,`$p_{k+1}$`)` *in the store. Depending on the case, one of the conditions hold:*

$$\min(p_1, \ldots, p_k) \geq \max(p_{k+1}, \ldots, p_n), \qquad \textit{or}$$

$$\min(p_1, \ldots, p_k) \geq p_{k+1} \geq \max(p_{k+2}, \ldots, p_n).$$

*Furthermore,*

$$\Pi \models \forall([\![Q]\!] \leftrightarrow ([\![Q_1, E_1]\!] \vee \cdots \vee [\![Q_n, E_n]\!]))$$

*and $p_i = F(E_i)$, $1 \leq i \leq n$. The* `printed_explain` *constraints hold minimal explanations and are added to the constraint store in a nondecreasing manner (i.e., never removed or changed) throughout the CHR derivation, and they appear in order of nondecreasing priority.* □

## 4 Probabilistic Abduction

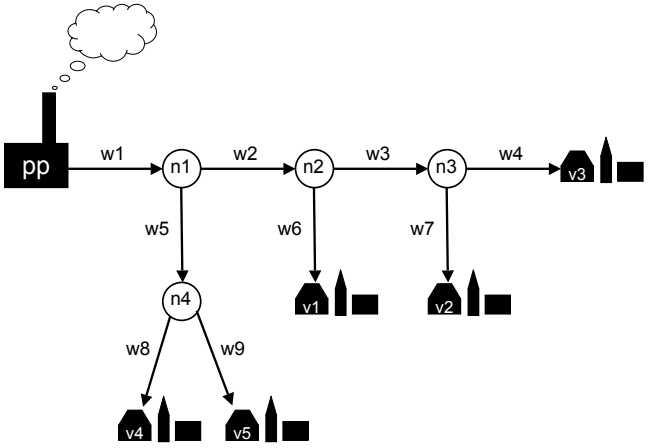
Consider an abductive logic program for which each abducible atom $A_i$ is seen as an independent random variable with two possible outcomes, *true* and *false*, with probability $p_i$ for *true*. As shown in details in [22], this gives rise to a probability distribution for the entire Herbrand universe and for explanations, and this distribution satisfies the conditions for being a priority function (definition 1).

In [22], we define a language of Probabilistic Abductive Logic Programs (PALP), in which abducibles are defined by declarations exemplified as follows.

```
abducible(a(_), 0.3).                                    (14)
```

The distribution defined by this declaration gives, e.g., that $P(\texttt{a(1)}) = P(\texttt{a(2)}) = 0.3$, and $P(\exists \texttt{a(X)}) = 1$.

We show here an example of a PALP and its implementation in CHR according to the recipe of section 3. We consider a power supply network which has one power plant `pp`, a number of directed wires $\texttt{w}_i$ and connecting nodes $\texttt{n}_i$, which may lead electricity to a collection of villages $\texttt{v}_i$. The overall structure is as follows.




Probabilistic abduction will be used to predict to most likely damages in the network given observations about which villages have electricity and which have not. As abducibles, we use `up/1` and `down/1` which apply to the power plant and the wires (for simplicity, the connecting nodes are assumed always to work). The network structure is represented by the following facts.

```
edge(w1, pp, n1).   edge(w4, n3, v3).   edge(w7, n3, v2).
edge(w2, n1, n2).   edge(w5, n1, n4).   edge(w8, n4, v4).    (15)
edge(w3, n2, n3).   edge(w6, n2, v1).   edge(w9, n4, v5).
```

The fact that a given point in the network has electricity, is described as follows.

```
haspower(pp):- up(pp).
haspower(N2):- edge(W,N1,N2), up(W), haspower(N1).           (16)
```

As no negation is supported, the program includes also clauses that simulate the negation of `haspower`.

```
hasnopower(pp):- down(pp).
hasnopower(N2):- edge(W,_,N2), down(W).                    (17)
hasnopower(N2):- edge(_,N1,N2), hasnopower(N1).
```

To express that `up/1` and `down/1` are each other's negation, we introduce an integrity constraint, and define probabilities that sum to one.

```
abducible(up(_), 0.9).
abducible(down(_), 0.1).                                   (18)
⊥:- up(X), down(X).
```

We assume that top-level goals are conjunctions of ground `haspower` and `hasnopower` atoms, which means that abducibles always are ground; this simplifies the implementation of the auxiliary predicates (with sorted lists for explanations) and allows some optimizations in the translating of clauses into CHR, by suppressing renaming of explanations and avoiding calls to `recalculate` following unifications. For example, the `haspower` predicates is translated as follows; recall that renaming is unnecessary for the `queue_explain` call that corresponds to the last clause, as the variables in the query are not referenced elsewhere.

```
step_explain( [haspower(N)|G], E, P) <=>
    rename([haspower(N)|G], [haspower(Nr1)|Gr1]),
    (Nr1=pp -> queue_explain([up(pp)|Gr1], E, P) ; true),      (19)
    queue_explain([edge(W2,N12,N),up(W2),haspower(N12)|G],E,P),
    select_best.
```

The implementation of the `extend` auxiliary (which is used when a new abducible is encountered) includes the checking of the integrity constraint; the full definition is as follows.

```
extend(A,E,P,AE,PAP):-
    insert_sorted(A,E,AE),
    abducible(A,PA),
    PAP is PA*P,
    \+ ic_fails(AE).
                                                           (20)
insert_sorted(X,[],[X]).
insert_sorted(X,[Y|Ys],[X,Y|Ys]):- X@<Y, !.
insert_sorted(X,[Y|Ys],[Y|Ys1]):- insert_sorted(X,Ys,Ys1).

ic_fails(E):- member(up(X),E), member(down(X),E).
```

The rest of the query interpreter contains no surprises. The following excerpt of a screen dialogue shows how the observation that no village have electricity is explained by the interpreter.

```
| ?- explain([hasnopower(v1), hasnopower(v2),
     hasnopower(v3), hasnopower(v4), hasnopower(v5)]).
Best solution: [down(w1)], Priority=0.1
Another solution? y
Best solution: [down(pp)], Priority =0.1
Another solution? y
Best solution: [down(w2),down(w5)],
Priority=0.010000000000000002
Another solution? y
Best solution: [down(w3),down(w5),down(w6)],
Priority=0.0010000000000000002
Another solution? y
Best solution: [down(w2),down(w8),down(w9)],
Priority=0.0010000000000000002
Another solution?
....
```
(21)

It appears that the intuitively two most reasonable hypotheses, namely that the power plant or the single wire connecting it with the rest of the network is down, are generated as the first ones with highest probability. Then follow combinations with lower and lower probability of different wires being down.

We refer to [22] giving implementations for the auxiliary predicates in two versions, for ground abducibles and for the general case.

## 5 Integration with Constraint Solving

Application of existing constraint solvers may provide effective ways of reducing the search space in abductive reasoning and may also help in producing more concise programs. We refer to such constraints as *external* as to distinguish them from the CHR constraints introduced above for the query interpreters. In principle, such external constraint solvers may also be written in CHR, the integration problem is the same.

In order to use external constraints with our query interpreters, we need a way to locate in the executions state, the constraints pending on given variables, so that we can copy them when new variants are produced of a running query in order to account for alternative program clauses. A traditional way to implement renaming without considering constraints is as follows,

```
rename(X,Y):- assert(aux(X)),retract(aux(Y)).
```
(22)

We exemplify here for the clp($Q$) constraint solver of SICStus Prolog [25, 26], how the renaming can be extended properly. It includes a predicate `projecting_assert`, which adds, in the body of a clause, the possible variables pending on its argument. We illustrate its use by an example; the curly brackets indicate the syntax for calling the constraint solver. Executing

```
{X=Y+Z}, projecting_assert(aux(p(X,Y,Z))).
```
(23)

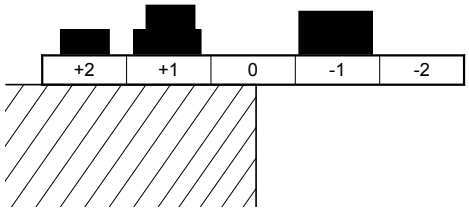
creates a clause equivalent with the following,

```
aux(p(X,Y,Z)):- {X=Y+Z}.
```
(24)

Calling this predicate with new arguments can set up the relevant constraints. The modified renaming predicate is as follows.

```
rename(X,Y):-
    assert(aux(X)),retract(aux(Y)),
    projecting_assert(aux(X)),
    aux(Y), retract((aux(_):- _)).
```
(25)

No more adjustments are needed to incorporate the use of this constraint solver. Consider as an example the following arrangement; the task is to place a number of objects on a board in the positions number $+2, \ldots, -2$ in such a way that the board does not tilt and fall to the ground. The priority function is designed with the purpose of achieving an arrangement with a minimum height for the given set of object, i.e., the maximum number $m$ of objects piled up in one position should be as small as possible; abducibles describe placements of objects and the priority of a given explanation is defined as $1/(m+1)$.




The PrioALP program is as follows; its translation into a query interpreter is straightforward according to the recipe of section 3.

```
abducible(pos(_object,_position)).
external({_}).

place_all:-
  pos([P1,P2,P3,P4,P5,P6,P7]),
  {P1*5+P2*7+P3*3+P4*7+P5*7+P6*12+P7*8 > 0},
  place([b1,b2,b3,b4,b5,b6,b7], [P1,P2,P3,P3,P5,P6,P7]).

pos([]).   pos([P|Ps]):- {P>=-2, P=<2}, pos(Ps).

place([],[]).

place([B|Bs],[PQ|Ps]):-
  member(P,[2,1,0,-1,-2]), {PQ=P},
  pos(B,P),
  place(Bs,Ps).
```
(26)

It considers the placement of objects `b1`, ..., `b7` having individual physical weights 5, 7, 3, 7, 7, 12 and 8. The purpose of the call {`PQ=P`} is to convert the integer $P$ into a representation of a rational number so that the constraint solver can work with it. Notice that a branch of computation is discarded as soon as the objects placed represent an overweight to the right in the picture, which is impossible to outbalance with the remaining objects.

```
| ?- explain([place_all]).
Best solution: [pos(b1,-2),pos(b2,2),pos(b3,2),pos(b4,-2),
                pos(b5,-1),pos(b6,1),pos(b7,0)],
Priority=0.33333
Another solution? y
...
```
(27)

On further requests follows a myriad of other solutions with the same priority, and far later solutions with priority 0.25, etc. So while this example illustrates how well an external constraint solver can be applied for pruning the search space as early as possible, it also indicates that a coarse priority function that does not distinguish possible solutions well, makes the search degenerate to a breath-first search. (A better function to try may be one that combines the height measuring with a priority of as much weight as possible to the left.)

## 6 Other Optimizations

Other optimizations have been considered which can be added to the query interpreters. For PrioALPs without integrity constraints, we can perform a pruning analogous to what happens in Dijkstra's shortest path algorithm [27]. Whenever we have two or more processes with the same remaining subgoal (e.g., for finding a path from the same intermediate node to the terminal node in the shortest path example), we keep only the best one; in CHR:

```
queue_explain(Q,E1,P1) \ queue_explain(Q,E2,P2) <=>
prority_less_than(P2,P1) | true.
```
(28)

When remaining subqueries are syntactically small (e.g., one call to a path predicate), this rule executes in an efficient way, and it will suppress the partial execution of some branches which are deemed not to become best in the end.

We mention also the possibility to apply simplified integrity constraints in specialized rules for each abducibles predicate. Simplification was suggested by [28] for database integrity checking; an unfolding of the theoretical foundations and a powerful method is given by [29]. We show here how this applies to the integrity checking shown in the code fragment (20) above. Typically simplification removes one order of magnitude (as in this example) or more.

```
step_explain([up(X)|G],E,P) <=>
    (member(down(X),E) -> true
     ;
     insert_sorted(up(X),E,E1), P1 is P*0.9,
     queue_explain(Q,E1,P1)),
    select_best.
```
(29)

This principle can be further extended with specialized treatment for clauses that produce more that one abducible.

**Acknowledgement:** This work is supported by the CONTROL project, funded by Danish Natural Science Research Council.

## References

1. Frühwirth, T.: Theory and practice of constraint handling rules, special issue on constraint logic programming. Journal of Logic Programming **37**(1–3) (October 1998) 95–138
2. Abdennadher, S., Christiansen, H.: An experimental CLP platform for integrity constraints and abduction. In: Proceedings of FQAS2000, Flexible Query Answering Systems: Advances in Soft Computing series, Physica-Verlag (Springer) (2000) 141–152
3. Christiansen, H.: CHR Grammars. Int'l Journal on Theory and Practice of Logic Programming **5**(4-5) (2005) 467–501
4. Christiansen, H., Dahl, V.: Logic grammars for diagnosis and repair. International Journal on Artificial Intelligence Tools **12**(3) (2003) 227–248
5. Christiansen, H., Dahl, V.: HYPROLOG: A new logic programming language with assumptions and abduction. In Gabbrielli, M., Gupta, G., eds.: ICLP. Volume 3668 of Lecture Notes in Computer Science., Springer (2005) 159–173
6. Christiansen, H., Martinenghi, D.: Symbolic constraints for meta-logic programming. Applied Artificial Intelligence **14**(4) (2000) 345–367
7. Henning Christiansen: Executable specifications for hypotheses-based reasoning with Prolog and Constraint Handling Rules. *Journal of Applied Logic*; to appear (2008)
8. Christiansen, H.: On the implementation of global abduction. In Inoue, K., Satoh, K., Toni, F., eds.: CLIMA VII. Volume 4371 of Lecture Notes in Computer Science., Springer (2006) 226–245
9. Satoh, K.: "All's well that ends well" - a proposal of global abduction. In Delgrande, J.P., Schaub, T., eds.: NMR. (2004) 360–367
10. Satoh, K.: An application of global abduction to an information agent which modifies a plan upon failure - preliminary report. In Leite, J.A., Torroni, P., eds.: CLIMA V. Volume 3487 of Lecture Notes in Computer Science., Springer (2004) 213–229
11. Frühwirth, T.W., Pierro, A.D., Wiklicky, H.: Probabilistic constraint handling rules. Electronic Notes in Theoretical Computer Science **76** (2002)
12. Charniak, E., Shimony, S.E.: Cost-based abduction and map explanation. Artificial Intelligence **66**(2) (1994) 345–374
13. Poole, D.: The independent choice logic for modelling multiple agents under uncertainty. Artif. Intell. **94**(1-2) (1997) 7–56
14. Stickel, M.E.: A prolog-like inference system for computing minimum-cost abductive explanations in natural-language interpretation. Annals of Mathematics and Artificial Intelligence **4** (1991) 89–105
15. Kakas, A., Michael, A., Mourlas, C.: ACLP: Abductive Constraint Logic Programming. Journal of Logic Programming **44** (2000) 129–177
16. Kakas, A.C., Nuffelen, B.V., Denecker, M.: A-system: Problem solving through abduction. In Nebel, B., ed.: IJCAI, Morgan Kaufmann (2001) 591–596

17. Endriss, U., Mancarella, P., Sadri, F., Terreni, G., Toni, F.: The ciff proof procedure for abductive logic programming with constraints. In Alferes, J.J., Leite, J.A., eds.: JELIA. Volume 3229 of Lecture Notes in Computer Science., Springer (2004) 31–43
18. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E.: The CHR-based implementation of a system for generation and confirmation of hypotheses. In Wolf, A., Frühwirth, T.W., Meister, M., eds.: W(C)LP. Volume 2005-01 of Ulmer Informatik-Berichte., Universität Ulm, Germany (2005) 111–122
19. Denecker, M., Kakas, A.C.: Abduction in logic programming. In Kakas, A.C., Sadri, F., eds.: Computational Logic: Logic Programming and Beyond. Volume 2407 of Lecture Notes in Computer Science., Springer (2002) 402–436
20. Kakas, A., Kowalski, R., Toni, F.: The role of abduction in logic programming. Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, Gabbay, D.M, Hogger, C.J., Robinson, J.A., (eds.), Oxford University Press (1998) 235–324
21. Pereira, L.M., Lopes, G., Dell'Acqua, P.: Pre and post preferences over abductive models. In Delgrande, J., ling, W.K., eds.: Multidisciplinary Workshop on Advances in Preference Handling (M-Pref'07) at 33rd Intl. Conf. on Very Large Data Bases (VLDB'07). (2007)
22. Henning Christiansen: Implementing Probabilistic Abductive Logic Programming with Constraint Handling Rules. To appear (2008)
23. Sneyers, J., Schrijvers, T., Demoen, B.: Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In Fink, M., Tompits, H., Woltran, S., eds.: WLP. Volume 1843-06-02 of INFSYS Research Report., Technische Universität Wien, Austria (2006) 182–191
24. Frühwirth, T.W., Holzbaur, C.: Source-to-source transformation for a class of expressive rules. In Buccafurri, F., ed.: APPIA-GULP-PRODE. (2003) 386–397
25. Christian Holzbaur: OFAI clp(q,r) Manual, Edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna (1995)
26. Swedish Institute of Computer Science: SICStus Prolog user's manual, Version 4.0.2. Most recent version available at `http://www.sics.se/isl` (2007)
27. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1**(4) (1959) 269–271
28. Nicolas, J.M.: Logic for improving integrity checking in relational data bases. Acta Informatica **18** (1982) 227–253
29. Christiansen, H., Martinenghi, D.: On simplification of database integrity constraints. Fundamenta Informatica **71**(4) (2006) 371–417