Proceedings of the Third Workshop on Constraint Handling Rules

Tom Schrijvers *Thom Frühwirth*

Report CW 452, June 2006

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Proceedings of the Third Workshop on Constraint Handling Rules

Tom Schrijvers *Thom Frühwirth*

Report CW 452, June 2006

Department of Computer Science, K.U.Leuven

Abstract

This book contains the proceedings of CHR 2006, the Third Workshop on Constraint Handling Rules, held at the occasion of ICALP 2006 in Venice (Italy) on July 9, 2006. The workshop 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.

Keywords : Constraint Handling Rules
CR Subject Classification : D.3.2, D.3.3, F.4.1

Preface

This book contains the Proceedings of CHR 2006, the Third Workshop on Constraint Handling Rules, held at the occasion of ICALP 2006 in Venice (Italy) on July 9, 2006.

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, sequents, 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 editions of the workshop were organized in May 2004 in Ulm (Germany) and October 2005 in Sitges (Spain). 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.

Eleven relevant papers were submitted to the workshop, all of which received at least two reviews and most even three. Two papers were rejected and another withdrawn. Eight excellent papers on CHR were accepted to the workshop.

We are privileged to have a distinguished invited speaker: Kazunori Ueda (Waseda University, Japan) talking about the LMNtal language, which is closely related to CHR.

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 ICALP 2006 organizers for hosting our workshop.

June 2006 Tom Schrijvers & Thom Frühwirth

Organization

Workshop Coordinators

Tom Schrijvers (Belgium) and Thom Frühwirth (Germany)

Program Committee

Alessandra Raffaeta, Universita Ca' Foscari Venezia, Italy

François Fages, INRIA Rocquencourt, France

Gregory J. Duck, Melbourne Universiy, Australia

Henning Christiansen, Roskilde University, Denmark

Jacques Robin, Universidade Federal de Pernambuco, Brazil

Martin Sulzmann, National University of Singapore, Singapore

Maurizio Gabbrielli, Universita di Bologna, Italy

Slim Abdennadher, German University in Cairo, Egypt

Thom Frühwirth, Universität Ulm, Germany

Tom Schrijvers, Katholieke Universiteit Leuven, Belgium

Additional Referees

Amira Thabet, German University in Cairo, Egypt

Hariolf Betz, Universität Ulm, Germany

Jon Sneyers, Katholieke Universiteit Leuven, Belgium

Leslie De Koninck, Katholieke Universiteit Leuven, Belgium

Noha Salem, German University in Cairo, Egypt

Paolo Baldan, Universita Ca' Foscari Venezia, Italy

Paolo Tacchella, Universita di Bologna, Italy

Peter Van Weert, Katholieke Universiteit Leuven, Belgium

Rémy Haemmerlé, INRIA Rocquencourt, France

Table of Contents

LMNtal as a Unifying Declarative Language

Kazunori Ueda[†], Norio Kato[‡], Koji Hara[†] and Ken Mizuno[‡]

[†]Dept. of Computer Science, Waseda University
[‡]Center for Verification and Semantics, National Institute of Advanced Industrial Science and Technology (AIST)
{ueda,n-kato,hara,mizuno}@ueda.info.waseda.ac.jp

Abstract. *LMNtal* (pronounced "*elemental*") is a simple language model based on hierarchical graph rewriting that uses logical variables to represent connectivity and membranes to represent hierarchy. LMNtal is an outcome of the attempt to unify constraint-based concurrency and Constraint Handling Rules (CHR), the two notable extensions to concurrent logic programming. LMNtal is intended to be a substrate language of various computational models, especially those addressing concurrency, mobility and multiset rewriting. Another important goal of LMNtal has been to put hierarchical graph rewriting into practice and demonstrate its versatility by designing and implementing a full-fledged, monolithic programming language. In this paper, we demonstrate the practical aspects of LMNtal using a number of examples taken from diverse areas of computer science. Also, we discuss the relationship between LMNtal and CHR, which exhibit both commonalities and differences in various respects.

1 Introduction

The development of the LMNtal language model has been motivated by two "grand challenges" in computational formalisms and programming languages. One is to have a computational model that unifies various paradigms of computation, especially those of concurrent computation and computation based on multiset rewriting. The other is to design and implement a practical programming language that covers a variety of computational platforms which are now developing towards both wide-area computation and nanoscale computation.

LMNtal is an outcome of the attempt to unify constraint-based concurrency (also known as concurrent constraint programming) [10] and (some ideas from) Constraint Handling Rules (CHR) [3], the two notable extensions to concurrent logic programming [9]. LMNtal can be viewed also as a multiset rewriting language equipped with links, where multisets are supported by the membrane construct that allows both nesting and mobility and links are represented by logical variables that essentially work as linear local names (i.e., local names occurring twice).

Despite its versatility, LMNtal is a surprisingly simple language and one can start using it with almost no background about programming or logic or

advanced mathematics. This is thanks to its close connection to diagrammatic representation of computational entities and the choice of the class of diagrams and reduction mechanisms to work with.

Since LMNtal was first designed in 2002 [11], it underwent the design review process from both theoretical and practical viewpoints. From a theoretical point of view, the major challenge has been to design the operational semantics in such a way that the interplay between graph structures formed by links and hierarchical structures formed by membranes (that may be crossed by links) is properly handled, which turned out to be quite subtle. From a practical point of view, the major challenge has been to build a full-fledged implementation of the newly designed language to provide designers with a constructive understanding of the language, to point out oversights in language design, to distinguish kernel constructs that require hard-wired support from those that can be implemented on top of the kernel, to accumulate programming experiences, and to identify language features not essential in theory but important in practice.

The LMNtal system, running on a Java platform and now available on the web[1], is the third of our attempts to implement the language. The purpose of this paper is to describe the features of LMNtal as a simple and versatile declarative language by means of various examples. The readers are referred to [12] on LMNtal as a computational model. All the examples in this paper have been tested on our LMNtal implementation.

2 Introductory Examples

2.1 Hierarchical Multiset Rewriting

The first series of examples is to demonstrate hierarchical multiset rewriting in LMNtal. Here we use LMNtal in an interactive mode.

```
$ lmntal
      LMNtal version 0.80.20060319
Type :h to see help.
Type :q to quit.

# 1,1,1, {1,1,1,1,1, {1,1,1}, (1,1:-2)}

1, 1, 1, {2, 2, 1, {1, 1, 1}, @601}

# {out,a,b,c}, d, {e,f}, ({out,$p[]} :- $p[]).

d, a, c, b, {f, e}, @603
```

Symbols starting with lowercase letters and numbers represent *atoms*, those starting with uppercase letters (not appearing yet) represent *links*, and braces

[1] http://www.ueda.info.waseda.ac.jp/lmntal/

represent *membranes*. Symbols starting with the dollar sign represent *process contexts*, and those starting with the at sign represent *rulesets*, which are (possibly compiled) sets of *rewrite rules*.

The first example is simple multiset rewriting (from two `1`'s to `2`), except that membranes are used to form hierarchical multisets. Rules local to membranes act only on those atoms in the same place in the membrane hierarchy. The symbol `@601` indicates a compiled ruleset obtained from the rule `(1,1:-2)`.

The second example shows that a rule can handle cells (i.e., atoms and cells enclosed by membranes) and change the membrane structure. The process context `$p[]` represents a local context of the membrane it belongs to and works as a wildcard. Thus the rule `({out,$p[]} :- $p[])` can be read as "remove the membrane containing the atom `out` and export its content." Notice that the order of atoms and cells are irrelevant because they form multisets.

2.2 List Processing

Next, we describe the use of links using list processing as an example.

The skeleton of a list can be represented, using `'.'` atoms (cons) and a `'[]'` atom (nil), as $\texttt{'.'}(A_1, X_1, X_0), \ldots, \texttt{'.'}(A_n, X_n, X_{n-1}), \texttt{'[]'}(X_n)$. Here, A_i is the link to the ith element and X_0 is the link to the whole list (from somebody else owning the list). This corresponds to a list formed by the constraints $X_0 = [A_1 \mid X_1]$, $\ldots$, $X_{n-1} = [A_n \mid X_n]$, $X_n = []$ in (constraint) logic programming languages, with the notable exception that an LMNtal list is a *resource* (i.e., entity with ownership) rather than a value. All links are point-to-point; link names occurring twice in an expression represent *local links* and those occurring once represent *free links* which are supposed to be connected to an atom outside the expression.

LMNtal links are non-directional like chemical bonds. The directionality of a list is determined by which arguments of atoms are connected together.

Two lists can be concatenated using the following two rules:

```
append(X0,Y,Z0), '[]'(X0) :- Y=Z0.
append(X0,Y,Z0), '.'(A,X,X0) :- '.'(A,Z,Z0), append(X,Y,Z).
```

(As above, each rule can be written in the period-terminated form as well as in the comma-separated form.) Figure 1 shows a graphical representation of the `append` program and its execution (`c`(ons) for `'.'` and `n`(il) for `'[]'`), where `b` is the consumer of the result of `append` and the atom `'='` autonomously disappears after connecting its arguments together.

LMNtal doesn't distinguish between predicate symbols (procedure names) and function symbols (constructors); they are equal in status, though a type system could be employed to distinguish between them.

LMNtal provides a simple, systematic and powerful *term abbreviation scheme*. We can exploit the fact that a local link name has exactly two occurrences and abbreviate $p(s_1, \ldots, s_m), q(\ldots, s_m, \ldots)$ to $q(\ldots, p(s_1, \ldots, s_{m-1}), \ldots)$. For instance, the list shown in the beginning of this section, with $n = 3$, can be abbreviated to $\texttt{'.'}(A_1, \texttt{'.'}(A_2, \texttt{'.'}(A_3, \texttt{[]})), X_0)$. This is structurally equivalent to $X_0 = Y$,

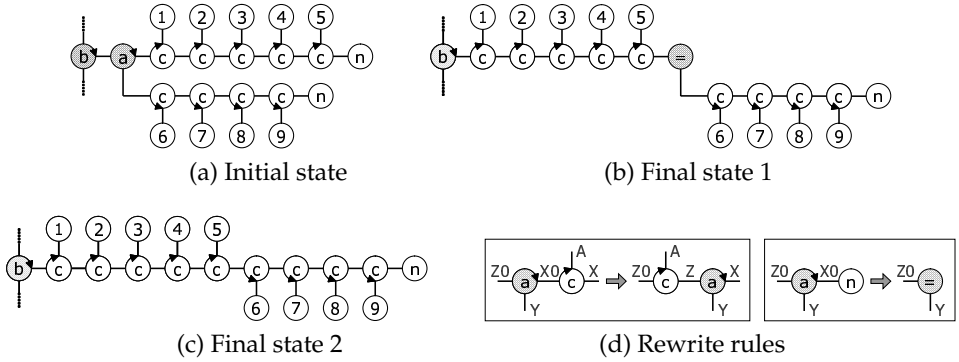


Fig. 1. List concatenation

‘.’(A_1,‘.’(A_2,‘.’(A_3, [])),Y) (see Section 3.2), and by eliminating Y using the scheme again we obtain X_0 =‘.’(A_1,‘.’(A_2,‘.’(A_3, []))) or X_0 = [A_1, A_2, A_3], following the Prolog convention.

Likewise, we write $p(\ldots, \{P\}, \ldots)$ to mean $p(\ldots, A, \ldots)$, $\{+A, P\}$, where P is a process (Section 3.1) and A is a link. The unary atom ‘+’ is used as the standard atom to terminate incoming links to a cell.

A notable consequence of the above scheme is that `f(5)`, `5(f)`, `f=5`, `5=f`, and `(5(X),f(X))` represent exactly the same thing, namely the unordered pair (or the diatomic molecule) of a unary `f` and a unary `5`.

Using abbreviation, the list concatenation program can be written in a (concurrent) logic programming form

```
append([],Y,Z) :- Y=Z.
append([A|X],Y,Z0) :- Z0=[A|Z], append(X,Y,Z)
```

and in the term rewriting form

```
Z= append([],Y) :- Z= Y.
Z= append([A|X],Y) :- Z= [A|append(X,Y)].
```

The above program resembles `append` in Interaction Nets [4]. Indeed, LMNtal generalizes Interaction Nets by removing the restriction to binary interaction and allowing hierarchical processes.

2.3 Numbers and Arithmetics

LMNtal supports numbers and arithmetics. However, for uniformity LMNtal treats every atom as resource and numbers are not exceptions. Accordingly, all numbers (integers and floats) are unary atoms and are connected to the “owner” of the number (which can be an arithmetic operator). LMNtal still allows 0-ary `8` and `3.14` as atoms but they are not supported by the arithmetic features of the language. In our LMNtal system, a ground expression autonomously evaluates to a number using a *system ruleset* implicitly built into every cell. For instance, the cell

(Process)	$P ::=$	$\mathbf{0}$	$\mid$	$p(X_1, \ldots, X_m)$	$\mid$	P, P	$\mid$	$\{P\}$	$\mid$ T `:-` T
(Process template)	$T ::=$	$\mathbf{0}$	$\mid$	$p(X_1, \ldots, X_m)$	$\mid$	T, T	$\mid$	$\{T\}$	$\mid$ T `:-` T
		$\mid$ `@`p	$\mid$	`$`p`[`$X_1, \ldots, X_m \mid A$`]`	$\mid$	$p(*X_1, \ldots, *X_n)$			
(Residual)	$A ::=$	`[]`	$\mid$	$*X$					

Fig. 2. Syntax of LMNtal

```
{n(1), n(2), n(3), n(4), n(5), (n(A),n(B):-n(A*B))}
```

evaluates nondeterministically to `{n(120), (n(A),n(B):-n(A*B))}`. LMNtal and our LMNtal implementation do not specify the strategy of multiset matching, but our implementation has a *shuffle* mode that randomizes the selection of atoms to see if the program may yield different results. For rules belonging to the same membrane, the textual order determines their natural priorities that can also be randomized by a runtime option.

3 The Core Language

We briefly describe the syntax and the semantics of LMNtal. For details omitted from here, the readers are referred to [12].

3.1 Syntax

The syntax of LMNtal is given in Figure 2, where two syntactic categories, *link names* (denoted by X) and *atom names* (denoted by p), are presupposed. The atom name `=` is reserved for atomic processes for link connection. The subject entities of LMNtal are called *processes*.

Intuitively, $\mathbf{0}$ is an inert process; $p(X_1, \ldots, X_m)$ $(m \geq 0)$ is an *atom* with m links; P, P is parallel composition called a *molecule*; $\{P\}$, a *cell*, is a process grouped by the membrane $\{\ \}$; and T `:-` T is a rewrite rule for processes.

An atom X`=`Y, called a *connector*, connects one side of the link X and one side of the link Y.

A process P must observe the following *link condition*: Each link name in P may occur *at most twice*. Furthermore, each link name of a rule must occur exactly twice. As usual, α-conversion can be used to avoid clashes of link names.

A *rule context*, `@`p, matches a (possibly empty) multiset of rules inside a membrane, while a *process context*, `$`p`[`$X_1, \ldots, X_m \mid A$`]` $(m \geq 0)$, matches processes other than rules inside a membrane. The argument of a process context specifies what links may or must occur free. When the residual A is `[]`, the argument is abbreviated to `[`$X_1, \ldots, X_m$`]` and means that the set of free links of `$`p must be exactly $\{X_1, \ldots, X_m\}$. When A is of the form $*X$ (called a *bundle*), it represents zero or more free links of the context that may occur in addition to the "must-occur" links $X_1, \ldots, X_m$.

$$\text{(E1)}\quad \mathbf{0}, P \equiv P \qquad \text{(E2)}\quad P, Q \equiv Q, P \qquad \text{(E3)}\quad P, (Q, R) \equiv (P, Q), R$$

$$\text{(E4)}\quad P \equiv P[Y/X] \qquad \text{if } X \text{ is a local link of } P$$

$$\text{(E5)}\quad P \equiv P' \Rightarrow P, Q \equiv P', Q \qquad \text{(E6)}\quad P \equiv P' \Rightarrow \{P\} \equiv \{P'\}$$

$$\text{(E7)}\quad X = X \equiv \mathbf{0} \qquad \text{(E8)}\quad X = Y \equiv Y = X$$

$$\text{(E9)}\quad X = Y,\ P \equiv P[Y/X] \qquad \text{if } P \text{ is an atom and } X \text{ occurs free in } P$$

$$\text{(E10)}\quad \{X = Y,\ P\} \equiv X = Y,\ \{P\} \qquad \text{if exactly one of } X \text{ and } Y \text{ occurs free in } P$$

$$\text{(R1)}\ \frac{P \longrightarrow P'}{P, Q \longrightarrow P', Q} \qquad \text{(R2)}\ \frac{P \longrightarrow P'}{\{P\} \longrightarrow \{P'\}} \qquad \text{(R3)}\ \frac{Q \equiv P \quad P \longrightarrow P' \quad P' \equiv Q'}{Q \longrightarrow Q'}$$

$$\text{(R4)}\quad \{X = Y, P\} \longrightarrow X = Y,\ \{P\} \qquad \text{if } X \text{ and } Y \text{ occur free in } \{X = Y, P\}$$

$$\text{(R5)}\quad X = Y,\ \{P\} \longrightarrow \{X = Y, P\} \qquad \text{if } X \text{ and } Y \text{ occur free in } P$$

$$\text{(R6)}\quad T\theta, (T :\!- U) \longrightarrow U\theta, (T :\!- U)$$

Fig. 3. Structural Congruence and Reduction Relation of LMNtal

The following examples will illustrate the role of process contexts. The LHS `{p(X)}` matches a cell exactly consisting of `p(X)` (`X` free), while the LHS `{p(X), $q[|*Y]}` matches a cell *containing* `p(X)` (`X` free) and the LHS `{p(X), $q[X|*Y]}` matches a cell *containing* `p(X)` (`X` local). The context receives the rest of the processes inside the membrane, and the bundle `*Y` is bound to the sequence of free links (order determined by the system) of the whole cell.

The final form, $p(*X_1, \ldots, *X_n)$ $(n > 0)$, represents an *aggregate* of n-ary atoms, which are to be connected to may-occur free links of process contexts.

Rewrite rules must observe syntactic conditions on possible occurrences of rules (inside the rules) and contexts, as well as on link names [12], all of which are for the use of contexts and aggregates to make sense.

3.2 Operational Semantics

The operational semantics of LMNtal (Figure 3) consists of two parts, namely structural congruence (E1)–(E10) and the reduction relation (R1)–(R6). Here, $[Y/X]$ is a *link substitution* that replaces X with Y.

(E1)–(E3) are the characterization of molecules as multisets. (E4) represents α-conversion of local link names. (E5)–(E6) are structural rules that make $\equiv$ a congruence. (E7) says that a self-absorbed loop is equivalent to $\mathbf{0}$, while (E8) expresses the symmetry of connectors. (E9)–(E10) are absorption/emission rules of connectors for atoms and cells, respectively.

Computation proceeds by rewriting processes using rules collocated in the same "place" of the nested membrane structure.

(R1)–(R3) are standard structural rules. (R1) says that reductions can proceed concurrently based on local reducibility conditions. Fine-grained concurrency of LMNtal originates from this rule. (R2) says that computation inside a membrane can proceed independently of the exterior of the membrane. For a cell to evolve autonomously, it must contain its own set of rules. Computation

of a cell containing no rules are to be controlled by rules outside the cell. (R3) incorporates structural congruence into the reduction relation.

(R4)–(R5) are the mobility rules of connectors across membranes. The central rule of LMNtal is (R6). The substitution θ is a mapping from process templates to processes and represents what process (or multiset of rules) has been received by each process context (or rule context), respectively, and what multiset of atoms each aggregate represents [12]. The simplest way of viewing rules with process/rule contexts and aggregates is to view them as *rule schemes* that represent sets of rules with no contexts or aggregates.

We can think of a subset of LMNtal, *Flat LMNtal*, that does not allow cell hierarchies (and accordingly, process contexts, rule contexts and aggregates). In Flat LMNtal, θ becomes unnecessary and (R6) is simplified to

$$\text{(R6}') \quad T, (T \text{ :- } U) \longrightarrow U, (T \text{ :- } U).$$

Matching between a process and the LHS of a rule under (R6′) should generally be done by α-converting the rule using (E4) and (R3). The whole resulting process, namely $U, (T \text{ :- } U)$ and its surrounding context, should observe the link condition, but this can always be achieved by α-converting T :- U before use so that the local link names in U won't cause name crashes with the context.

3.3 Extension: Guard and Typed Process Contexts

Our LMNtal system extends the above core language by featuring the notion of *guards* and *typed process contexts*. Guards are to express conditional rewrite rules, but the question is what should constitute conditions in our setting of graph rewriting. Suppose we have a multiset of integers expressed as

```
n(3), n(-5), n(101), n(72), n(18), n(47), n(11).
```

and want to find the maximum value of them by deleting others. One can achieve this using a one-liner

```
n($i), n($j) :- $i =< $j | n($j).
```

which is an abbreviation of

```
n(I), $i[I], n(J), $j[J] :- $i =< $j | n(K), $j[K].
```

Here, the process contexts don't appear in membranes that would delimit the contexts, but instead appear as operands of the guard test `$i=<$j` which constrains its arguments to unary atoms. The guard test checks if the two unary atoms are both integers and the second one is not smaller.

Other guard tests available include `int($i)` to ensure that `$i` is an integer, `unary($u)` to ensure that `$u` is a unary atom, and `ground($g)` to capture a minimal (and hence connected) graph structure with exactly one free link. Note that '`=<`', `unary`, and `ground` can all be regarded as *type constraints*; `int` is a subtype of `unary` which in turn is a subtype of `ground`, and '`=<`' ia s subtype of the product `int` $\times$ `int`.

Graph structures received by typed process contexts and checked by guard tests can be copied or discarded freely. Thanks to this mechanism, many of concurrent logic programs ever written run as LMNtal programs with very minor modifications, where the *ask* operation (the synchronization primitive) is naturally replaced by graph matching.

LMNtal features another important guard test, `uniq`$(X_1, \ldots, X_n)$. This test succeeds if (i) each X_i is connected to a (connected) graph with no free links other than X_i, and (ii) the rule has not been applied to the same tuple of graphs before. As a special case of $n = 0$, `uniq` succeeds if the rule in question has not been used before. The `uniq()` test, which was inspired by CHR's propagation rules, is a general tool for avoiding infinite application of rules whose RHS is a super(multi)set of the LHS.

4 Ideas Behind the Language Design

4.1 Basic Ingredients

The "four elements" of LMNtal are *logical links, multisets (or membranes), nested nodes, and transformation*.

Links, which are represented by linear local names, are used to represent both one-to-one communication channels between logically neighboring processes and logical neighborhood relations between data cells. LMNtal viewed as a process calculus is different from many other process calculi (or more precisely, it is a special case of them) in that a message sent through a link changes the identity of the link and that links are always private in the sense that the third party cannot access them. The conception of logical links originate from logical variables in logic programming, but again they are the special case of logical variables in that LMNtal has no notion of *instantiating* a link variable to a value; it just retains the notion of *fusing* two variables.

The notion of multisets can be found in diverse computational models including classical Petri Nets and Production Systems. However, not many of them feature multisets as first-class citizens, and the essence of membranes is to give hierarchical structures to the systems of multisets. Models and languages featuring membranes include the Chemical Abstract Machine [1], mobile ambients [2], P-systems [6], bigraphical reactive systems [5], LMNtal, and the Kell Calculus [7].

4.2 Uses of Membranes: The π-calculus Example

Membranes play many roles in LMNtal programming. First, they are used to represent records or feature structures. Second, they can encapsulate rules and delimit their scope of effect. Third, they can protect processes from the rewrite rules that would otherwise act on the processes.

The following example that encodes the communication mechanism of the asynchronous π-calculus illustrates the first and the third uses of membranes. The rules are prefixed by rule names here.

```
snd@@ snd({$y[|*V]},X) :- {$y[|*V], m(X)}.
get@@ get({m(X),$y[|*V]},Z), {$body[Z|*V]} :- {$y[|*V]}, $body[X|*V].
cp@@ {name(N),$p[N|*Y],+Z}, Z=cp(Z0,Z1) :- {name(N),$p[N|*Y],+Z0,+Z1}.
rm@@ {name(N),$p[N|*Y],+Z}, Z=rm :- {name(N),$p[N|*Y]}.
```

Here, a π-calculus name is represented by a cell containing the `name()` attribute and referenced by incident links (each marked by the '+' atom). The cell also works as a message buffer where the links marked by the `m` atom are connected to outstanding messages. The first rule gives the semantics of sending x to y, while the second rule gives the semantics of receiving a message x from y and substitute it for the formal name z. The last two rules, `cp` for copy and `rm` for remove, are used when a formal name is used more than once or not used at all. For instance, a π process $(a(z).b(y).\overline{z}\langle y\rangle) \mid \overline{a}\langle c\rangle \mid \overline{b}\langle d\rangle$ is encoded as

```
(get(A0,Z), {get(B0,Y), {snd(Z,Y)}}).
snd(A1,C).
snd(B1,D).
{name(a),+A0,+A1}, {name(b),+B0,+B1}, {name(c),+C}, {name(d),+D}.
```

which reduces to

```
{name(a)}, {name(b)}, {name(c),m(_78)}, {name(d),'+'(_78)}, @601
```

meaning that the channel `c` contains an outstanding message `d`, while `a` and `b` do not hold any messages nor are referenced any more. Note the use of membranes in the encoding of $(a(z).b(y).\overline{z}\langle y\rangle)$; prefixed processes are protected by membranes until the `get` rule removes them.

4.3 Logical Interpretation

LMNtal processes are designed to allow diagrammatic representation, but at the same time processes and rules allow logical interpretation. We focus on Flat LMNtal (LMNtal without membranes) due to space limitation.

A (Flat) LMNtal process is a conjunction of atoms where local links are interpreted as existentially quantified variables. First-order logic with equality (to handle connectors) works as the underlying logic in many cases. For programs dealing with multisets (i.e., conjunction of two or more identical formulae), however, we should drop weakening and contraction from the logic, which results in a (tiny) fragment of linear logic. Programs with don't-care nondeterminism are another example in which linear logic interpretation would be more appropriate. In the following, we focus on the classical case.

A rule T :- U is interpreted as $\forall(\exists T \Leftrightarrow \exists U)$, where T and U are conjunctions of atoms, the two $\exists$'s quantify the *local* links of T and U, respectively, and the leftmost $\forall$ quantifies the *free* links of T and U (each occurring once in T and once in U). Thus a LMNtal rule is not a clause in the classical sense, but it is easy to see that the rewrite rule (R6'), $\exists T \wedge \forall(\exists T \Leftrightarrow \exists U) \longrightarrow \exists U \wedge \forall(\exists T \Leftrightarrow \exists U)$, is a sound reasoning. (Note that the quantifier of the redex $\exists T$ works on local links of T only; free links in T are left free.) For instance, under the `append` program,

the molecule `append([a,b],[c],X),answer(X)` reduces to `answer([a,b,c])`, which is a sound deduction.

Note the symmetry of our logical reading of rules. This means that the initial state `append([a,b],[c],X),answer(X)` can be deduced from `answer([a,b,c])` as well. Computationally, the base case of `append`, if reversed, would cause divergence because it would match any link in the current configuration. However, there is a class of reversible programs whose initial states can be restored from the final states by the backward application of LMNtal rules, an interesting topic of future study.

Our logical interpretation is to be contrasted with that of CHR where its simplification rule T`<=>`U is interpreted as $\forall(T \Leftrightarrow \exists U)$ [3]. The difference comes from the purpose of the languages (in general) and the roles of the variables (technically) that will be discussed below.

4.4 Relation to CHR

LMNtal rules without membranes resemble simplification rules of CHR. Indeed, Flat LMNtal could be thought of as a linear fragment of CHR. However, we find LMNtal and CHR quite different and rather complementary. First, CHR was developed for glass-box constraint programming, while LMNtal was developed as a declarative concurrent language with powerful data structures (hierarchical graphs that subsume first-class multisets). Second, CHR is to be used with some platform language (such as Prolog and Java) while LMNtal was developed as a monolithic, stand-alone language. Third, LMNtal comes with membranes that can be used not only as data structures but also as control structures, which are essential for a stand-alone language. Fourth, CHR resembles Prolog in the use of constructors and logical variables, while LMNtal is constructor-free and restricts variables to linear ones representing connectivity. Fifth, computation in LMNtal is not necessarily intended to be confluent because its intended applications include concurrent programming.

Nevertheless, a logic variable (possibly with attributes as in some Prolog implementations) can be encoded using membranes. Furthermore, the `uniq` guard test (Section 3.3) has been used successfully to encode propagation rules of CHR.

5 Overview of the Implementation

The current version of our LMNtal system consists of 43,000 lines of Java code including programming environments. Both the `lmntal` and `lmnc` commands invoke the compiler to generate dedicated intermediate code. The `lmntal` command executes it interpretively, while `lmnc` translates it further into Java code packed into a Java archive file, which can be executed by the `lmnr` command. Several subtleties, mostly resulting from the hierarchical nature of the language, have been identified and resolved in the course of design reviews.

LMNtal is a fine-grained concurrent language, but how to implement it correctly and efficiently is far from obvious for the following two reasons. First, it features both connectivity (links) and hierarchy (membranes) in such a way that links may connect remote atoms across one or more membranes. Second, different rulesets belonging to different membranes may attempt to rewrite the same process competitively; for instance, a process located at some place in the membrane hierarchy may be manipulated by both local and non-local rules. In addition, since LMNtal features interface to Java and Java allows users to create threads—explicitly or implicitly by using GUI's—, being able to control asynchronous execution is an important requisite. The major challenge at this stage of language development has been to establish a correct implementation scheme amenable to asynchronous rewriting by multiple tasks (though we have already implemented several, mostly intra-rule, optimization techniques including dependency-directed backtracking of multiset matching, reuse of atoms and links upon reduction, etc.).

One of our decisions was to let a link cross a membrane via two "immigration" proxies, one inside the membrane and the other outside. For instance, the standard internal representation of the configuration `{a(X)},b(X)` is

```
{a(X0), $in(X1,X0)}, $out(X1,X2), b(X2)
```

as one can see by raising the verbosity level of the system. Semantically, these proxies are just connectors (`=`) sticking to the membrane.

These proxies enable concurrent rewriting of subgraphs belonging to different places of the membrane structure. Having proxies both inside and outside the membrane also eases the implementation of (R4) and (R5) by letting them react autonomously when two `$in`'s or two `$out`'s are tied together.

6 More Examples

The programs we have successfully expressed using LMNtal are quite diverse, including the pure and call-by-name lambda calculi, the synchronous and the asynchronous π-calculus (see Section 4.1 for the asynchronous case), the ambient calculus, bigraphs and their composition [5], bottom-up parsers and unparsers, calculators with GUI, fullerenes (as examples of highly-connected graph structures), to name a few.

6.1 The Lambda Calculus

The lambda calculus based on graph reduction can be elegantly encoded in LMNtal. Here we give a nondeterministic version (Fig. 4).

This is a simplified version of the encoding into Interaction Nets by Sinot [8]. The first rule captures the essence of the lambda calculus, β-reduction, while all the other rules are to handle nonlinear (i.e., $\neq 1$) use of variables by copying or removing graph structures. The key idea of [8] was to use two different atoms, `cp` and `dp`, to control graph copying.

For instance, the Church numeral 2 ($\lambda fx.f(fx)$) is encoded as

```
beta@@ H=apply(lambda(A, B), C) :- H=B, A=C.

l_c@@ lambda(A,B)=cp(C,D) :- C=lambda(E,F), D=lambda(G,H), A=dp(E,G), B=dp(F,H).
a_c@@  apply(A,B)=cp(C,D) :- C= apply(E,F), D= apply(G,H), A=cp(E,G), B=cp(F,H).
l_d@@ lambda(A,B)=dp(C,D) :- C=lambda(E,F), D=lambda(G,H), A=dp(E,G), B=dp(F,H).
a_d@@  apply(A,B)=dp(C,D) :- C= apply(E,F), D= apply(G,H), A=dp(E,G), B=dp(F,H).
l_r@@ lambda(A,B)=rm :- A=rm, B=rm.
a_r@@  apply(A,B)=rm :- A=rm, B=rm.
c_r@@     cp(A,B)=rm :- A=rm, B=rm.
d_r@@     dp(A,B)=rm :- A=rm, B=rm.
r_r@@          rm=rm :- .
d_d@@ dp(A,B)=dp(C,D) :- A=C, B=D.
c_d@@ cp(A,B)=dp(C,D) :- C=cp(E,F), D=cp(G,H), A=dp(E,G), B=dp(F,H).
u_c@@ U=cp(A,B) :- unary(U) | A=U, B=U.
u_d@@ U=dp(A,B) :- unary(U) | A=U, B=U.
u_r@@ U=rm      :- unary(U) | .
```

Fig. 4. The lambda calculus, nondeterministic version

```
lambda(cp(F0,F1),lambda(X,apply(F0,apply(F1,X))),Result).
```

and

```
N=n(2) :- N=lambda(cp(F0,F1), lambda(X, apply(F0,apply(F1,X)))).
N=n(3) :- N=lambda(cp(F0,cp(F1,F2)), lambda(X,
                apply(F0,apply(F1,apply(F2,X))))).
res=apply(apply(apply(n(2), n(3)), s), 0).
H=apply(s, I) :- int(I) | H=I+1.
```

evaluates to a molecule `res=9` (plus the initial rules), where the readers are reminded that exponentiation of Church numerals is encoded as $\lambda mn.nm$. As illustrated above, the encoding in LMNtal allows some free names (`s` in this example) to be "evaluated" by rules given by the user (δ-reduction).

We have also encoded the call-by-name lambda calculus, and ran recursive functions expressed using the fixpoint (**Y**) combinator.

6.2 Highly Connected Graph Structures

In most declarative languages (except logic programming languages featuring unification over rational terms), cyclic or highly connected data structures are harder to manipulate than lists and trees. To demonstrate that this is not the case with LMNtal, we give a simple program to build a fullerene (C_{60}) structure consisting of only two rules and two initial atoms:

```
dome(L0,L1,L2,L3,L4,L5,L6,L7,L8,L9) :-
    p(T0,T1,T2,T3,T4), p(L0,L1,H0,T0,H4), p(L2,L3,H1,T1,H0),
    p(L4,L5,H2,T2,H1), p(L6,L7,H3,T3,H2), p(L8,L9,H4,T4,H3).
dome(E0,E1,E2,E3,E4,E5,E6,E7,E8,E9).  /* top half */
```

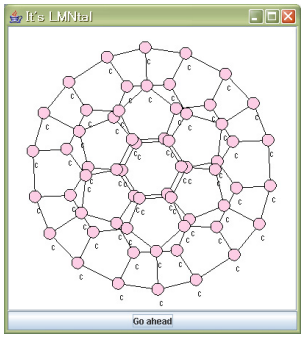


Fig. 5. The C_{60} structure

```
{ module(io).
  ...
  io.input(Message, X) :- [:/*inline*/
    String s = javax.swing.JOptionPane.showInputDialog(null, me.nth(0));
    me.setName(``done'');
    me.nthAtom(0).setName(s);
    :](Message, X).
  ...
}
```

Fig. 6. The `io` module

```
dome(E0,E9,E8,E7,E6,E5,E4,E3,E2,E1).  /* bottom half */

p(L0,L1,L2,L3,L4) :- X=c(L0,c(L1,c(L2,c(L3,c(L4,X))))).
```

The first rule and the next two initial atoms build a icosahedron (polyhedron with 20 triangles) by sewing up two five-triangle domes, and the final rule turns it into a fullerene structure. Figure 5 shows the graph structure rendered using the visualization option (-g) of the LMNtal system.

6.3 Modules and Foreign-Language Interface

The system comes with modules and foreign-language interface, both extremely important for developing applications. For instance, the `io` module, coming as a standard library, starts with the definition of Fig. 6.

A module is just a membrane containing the module name declaration and a set of rules. When a(n extended) name of the form *modulename.atomname* is mentioned in some membrane, the ruleset belonging to the module is implicitly imported to that membrane.

This example also shows the use of Java code in LMNtal. The Java code appears as a special atom, quoted by `[:` and `:]` and starting with `/*inline*/`, which is called an *inline execution atom*. The code is expanded in the translated

Java code and executed in the final phase of rule application, so that the code can access graph structures built by the RHS of the rule. The special variable `me` refers to the atom (which is the Java code being executed) and `mem` (not used in the example) refers to the membrane it belongs to. The *i*th argument given to the inline code can be accessed as `me.nth(`*i*`)`. Using these mechanisms, one can manipulate LMNtal's graph structures from within the inline atom. For instance, executing `result=io.input("Hello")` will show a pop-up window saying `"Hello"`, and typing in `"World!"` into the text field will cause the the whole molecule to evolve into `result=done("World!")`.

One can also define Java classes used by inline code using an *inline declaration atom*, a quoted atom starting with `/*inline_define*/`. The foreign-language interface greatly facilitated the development of the LMNtal system because one could easily provide and test new features (arrays, sockets, window toolkits, etc.) before or without hard-coding them into the runtime system.

7 Concluding Remarks

We have presented a concise declarative language LMNtal with diverse program examples. The main contribution of the work is that we have succeeded in putting hierarchical graph rewriting into practice and demonstrated its versatility. We have also shown the logical interpretation of LMNtal computation. LMNtal inherits some of the ideas from constraint-based concurrency and CHR, but cell hierarchy that allows trans-membrane links and local rulesets required us to develop a new implementation technique almost from scratch.

CHR is another multiset rewriting language that features logical variables. While Flat LMNtal could be thought of as a linear fragment of CHR, they exhibit differences as well as commonalities in various respects including the use of logical variables, the control structure, and principal applications. Still, we believe that their commonalities call for the cross-fertilization of the ideas, results and experiences we have accumulated.

Parallel and distributed implementations of LMNtal are both underway, building upon the asynchronous execution scheme we have developed.

LMNtal opens up many interesting research issues. One of the most important issues in language design and implementation is to equip it with useful type systems. We believe that many useful properties, for instance shapes formed by processes and the directionality of links (i.e., whether links can be implemented as one-way pointers), can be guaranteed statically using type systems and enable aggressive compiler optimization. Another important topic is to design and implement appropriate constructs for (don't-know) nondeterministic computation. Although LMNtal started as a concurrent language, it is now addressing diverse applications including those involving (don't-know) nondeterminism (e.g., verification), and backtracking or exhaustive search for possible reduction paths is becoming much more important than we had expected. We have finished a prototype implementation of backtracking and ex-

haustive search for Flat LMNtal. Extending it to deal with hierarchical graphs is far from obvious, and is a challenging topic of future research.

Acknowledgments

The authors are indebted to the past and current members of the LMNtal project, particularly Takahiko Nagata, Shingo Yajima, Shintaro Kudo and Ken Sakurai for the development of the ideas described here and the code they contributed. This work is partially supported by Grant-In-Aid for Scientific Research ((B)(2) 16300009, Priority Areas (C)(2)13324050 and (B)(2)14085205), MEXT and JSPS.

References

1. Berry, G. and Boudol, G., The Chemical Abstract Machine. In *Proc. POPL'90*, ACM, pp. 81–94.
2. Cardelli, L. and Gordon, A. D. : Mobile Ambients, in *Foundations of Software Science and Computational Structures*, Nivat, M. (ed.), LNCS 1378, Springer, 1998, pp. 140–155.
3. Frühwirth, T., Theory and Practice of Constraint Handling Rules. *J. Logic Programming*, Vol. 37, No. 1–3 (1998), pp. 95–138.
4. Lafont, Y., Interaction Nets. In *Proc. POPL'90*, ACM, pp. 95–108.
5. Milner, R., Bigraphical Reactive Systems. In *Proc. CONCUR 2001*, LNCS 2154, Springer, 2001, pp. 16–35.
6. Păun, Gh., Computing with Membranes. *J. Comput. Syst. Sci.*, Vol. 61, No. 1 (2000), pp. 108–143.
7. Schmitt, A. and Stefani, J.-B., The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Proc. Int. Workshop on Global Computing.*, LNCS 3267, Springer, 2005, pp. 146–178.
8. Sinot, F.-R., Call-by-Name and Call-by-Value as Token-Passing Interaction Nets. In *Proc. TLCA 2005*, LNCS 3461, Springer, 2005, pp. 386–400,
9. Ueda, K., Concurrent Logic/Constraint Programming: The Next 10 Years. In *The Logic Programming Paradigm: A 25-Year Perspective*, Apt, K. R., Marek, V. W., Truszczynski M., and Warren D. S. (eds.), Springer-Verlag, 1999, pp. 53–71.
10. Ueda, K., Resource-Passing Concurrent Programming. In *Proc. TACS 2001*, LNCS 2215, Springer, 2001, pp. 95–126.
11. Ueda, K. and Kato, N., Programming with Logical Links: Design of the LMNtal Language. In *Proc. Third Asian Workshop on Programming Languages and Systems (APLAS 2002)*, 2002, pp. 115–126.
12. Ueda, K. and Kato, N., LMNtal: a language model with links and membranes. In *Proc. Fifth Int. Workshop on Membrane Computing (WMC 2004)*, LNCS 3365, Springer, 2005, pp. 110–125.

Towards Agent Programming in CHR

Edmund S. L. Lam and Martin Sulzmann

School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
{lamsoonl,sulzmann}@comp.nus.edu.sg

Abstract. We investigate an approach to the design and implementation of linear logic based agent systems via the linear logic semantics of Constraint Handling Rules (CHR). The intuition behind our approach is simple: Linear logic provides strong logical foundations to reason, verify and specify agent systems beyond the limitations of classical logics, while with CHR, one can implement and analyse agent systems in a concise and compact manner by executable inference rules. We discuss necessary refinements of the CHR semantics to allow for sequential computations of actions and the verification of action determinism. Our approach can possibly provide a seemless integration of the formal specification and implementation of agent programs via CHRs.

1 Introduction

The idea of autonomous intelligent agent in programming has become increasing prevalent since its introduction in the early days of Artifical Intelligence. The concept of intelligent agents have opened new possibilities to computerization, allowing complex tasks previously thought to be beyond the computational limits of programming, to be mechanized. From advanced web service applications, to telemetry control systems for space exploration vessels (NASA's Deep Space 1, DS1 [MNPW98]), agent systems have been successfully deployed in a wide range of applications. Yet in spite of active research and progress into formal methods for agent specification, development of agent systems is still a difficult and tedious process because of the 'gap' between the abstract agent specifications and the concrete implementations.

In this paper, we explore an approach to the design and implementation of linear logic based agent systems via Constraint Handling Rules [Frü95] (CHR). Linear logic [Gir95] provides strong logical foundations to reason, verify and specify agent systems beyond the limitations of classical logics, thus avoiding the frame problem [MH69] entirely. With CHR one can implement and analyse agent systems in a concise and compact manner by executable inference rules. Recent works in [BF05] have shown that CHRs have a natural linear logic declarative semantics, establishing a long awaited bridge between the two worlds. Hence we can *directly* program linear logic agents in CHRs. One major advantage is that many agent domain properties can be stated and checked via the executable CHR operational semantics. Thus, CHR encodings of agent systems can support an effective and seemless integratation of specification and implementation. Specifically, we make the following contributions:

– We explore the benefit of CHRs to describe linear logic agents (Section 2). For example CHRs provide a natural and direct solution to the frame problem.

– We introduce an agent oriented refinement of CHR semantics, Monadic Action CHR Semantics, to support action sequencing (Section 3).
– We explore the verification of action determinism via a refined confluence test with respect to the domain invariants of the agent domain (Section 4).

In Section 3 we provide background material on CHRs as well as discuss the various issues of our approach. We also highlight the necessary refinements to the CHR semantics. Following this, in Section 4 we discuss action determinism and confluence. We conclude in Section 5 where we also discuss related works.

2 A Motivating Example

We consider a simple example called the *Block world*, which is a commonly used example to illustrate planning and domain representation problems in artifical intelligence. Block world consist of several blocks stacked on several tables and a robotic arm capable of picking up and putting down blocks one at a time. We consider a simple instance of Block world consisting of 3 blocks($B1, B2, B3 \in Blocks$), 3 tables(Labelled $T1, T2, T3 \in Tables$) and the robotic arm. To describe the the block world domain, we define four fluents (properties of the domain); $On(x, y)$ states that a block x is on block or table y, $Clear(x)$ states that nothing is stacked on x, $Empty$ states that the robot arm is empty and $Holds(x)$ says that the robot arm is holding x. A state is represented by a set of fluents described above. Block world has 2 legal actions described by the following.

– ***get(x)***: Provided the robotic arm is empty, block x is clear and x is on some y, the arm can pick up x, causing x to be no longer clear and on y, with x in the arm and y clear.
– ***putOn(x,y)***: Block x can be placed on y provided that x is held in the robotic arm, and y is clear. This makes the robotic arm empty and y no longer clear, with x on y.

To complete the informal specification, we also require the following domain invariants, labelled $\mathcal{I}^{Bw}$ 1 to 5.

– $\mathcal{I}_1^{Bw}$: Robotic arm cannot be both empty and holding something.
– $\mathcal{I}_2^{Bw}$: Objects cannot be both clear and have something on it.
– $\mathcal{I}_3^{Bw}$: Objects can only be stacked on and with at most one other object.
– $\mathcal{I}_4^{Bw}$: Robotic arm can hold at most one object.
– $\mathcal{I}_5^{Bw}$: Objects held by robotic arm cannot be clear or stacked on or with other objects.

Notice that if we try to represent actions naively in classical logic, we get an inconsistency as they contradict the domain invariants. Propositions, in classical logic, are viewed as 'truths' and once asserted, they are persistently valid. Situation Calculus [MH69] provides a solution to the representation of dynamically changing domains in classical logics. In situation calculus, we parameterize each fluent' additionally with a *situation*. Situations represent the sequence of actions that has lead to it's truth. For example, the fluent $Empty(s_1)$ where $s_1 = putOn(B1, T3) : get(B1) : S_0$ (we use the operator ':' as a standard list concatenation operator) simply asserts that the robotic arm is empty in the

state after the actions $get(B1)$ followed by $putOn(B1, T3)$ are executed from the initial situation S_0. Situations act as 'time tags' that distinguishes fluents of different states. With our fluents now parameterized by the situation, we are able to formulate the block world actions in first order logic:

$$
\begin{array}{ll}
(get) & \forall x, y, S. \\
& Empty(S) \wedge Clear(x, S) \wedge On(x, y, S) \supset Holds(x, get(x) : S) \wedge Clear(y, get(x) : S) \\
(putOn) & \forall x, y, S. \\
& Holds(x, S) \wedge Clear(y, S) \supset Empty(putOn(x, y) : S) \wedge On(x, y, putOn(x, y) : S) \wedge \\
& \qquad\qquad Clear(x, putOn(x, y) : S)
\end{array}
$$

It would seems that we can correctly represent actions simply by introducing situation 'time tags'. However we have also unwittingly introduced a new problem, known as the *Frame problem*.

2.1 The Frame Problem

Let's consider the situation calculus representation presented in the previous section. Suppose from the initial state $\Delta_0 = Empty(S_0) \wedge On(B1, B2, S_0) \wedge On(B2, B3, S_0) \wedge On(B3, T1, S_0) \wedge Clear(B1, S_0) \wedge Clear(T2, S_0) \wedge Clear(T3, S_0)$, the robotic arm executes $get(B1)$. Thus, from Δ_0 using the formula $get(B1)$, we can entail that $Holds(B1, S_1) \wedge Clear(B2, S_1)$ where $S_1 = get(B1) : S_0$. This yields the next state, $\Delta_1 = \Delta_0 \wedge Holds(B1, get(B1) : S_0) \wedge Clear(B2, get(B1) : S_0)$, which is unfortunately still incomplete, since we should also have $Clear(T2, S_1)$, $Clear(T3, S_1)$, $On(B2, B3, S_1)$ and $On(B3, T1, S_1)$. The formulation of the actions given above do not propagate the non-effects (non-effects refers to fluents that are not involved in the action, which should remain the way they were in the next situation). The problem of explicitly representing these non-effects of actions is known as the frame problem [MH69].

There are many proposed solutions to the frame problem [BMR95,Thi98]. The *successor state axioms* [BMR95] is one such solution. In [Thi98], Fluent calculus is proposed, solving the frame problem with *state update axioms*, which describe actions via explicit state representation. No doubt each of these solutions solves the frame problem, yet the cost in complexity and effort would often make any attempt in specifying practical dynamic agent systems in such classical logic based calculi extremely tedious, time consuming and at times difficult to read.

2.2 Block World via Linear Logic & CHR

Linear logics [Gir95] is an extension of classical logics. In linear logics, propositions are treated as expendible resources rather than irrefutable truths. This corresponds more to properties of a dynamically changing physical domain. The block world actions can be accurately represented by the following *linear implications*:

$$
\begin{array}{rl}
(get) & !(\forall x, y.\ Empty \otimes Clear(x) \otimes On(x, y) \multimap Holds(x) \otimes Clear(y)) \\
(putOn) & !(\forall x, y.\ Holds(x) \otimes Clear(y) \multimap Empty \otimes On(x, y))
\end{array}
$$

This linear logic interpretation of the block world is consistent and correctly represents the intended dynamics of the block world. Informally speaking, the implication ($\multimap$) says replace the left-hand-side conjuncts ($\otimes$) with the right-hand-side. The application of *get* $B1$ will result in the removal of $Empty$ (among others) before introduction of $Holds(B1)$ (among others), thus avoiding inconsistency.

The semantics of linear logic implications are remarkably similar to the operational semantics of the CHR simplification rule. Works in [BF05] have concretized this link between CHR and linear logic, suggesting that we can *directly* encode such agent systems as a CHR program. Here is the straightforward encoding via CHRs.

$$\begin{array}{rcl} get & @ & Empty, Clear(x), On(x,y) \Longleftrightarrow Holds(x), Clear(y) \\ putOn & @ & Holds(x), Clear(y) \Longleftrightarrow Empty, On(x,y), Clear(x) \end{array}$$

Each action in the block world is associated with a CHR rule to which we refer to as an *action rule*. A state in block world is simply represented by a CHR execution state. Like their linear logic counterparts, these direct CHR translations of the actions naturally solve the frame problem. Yet they are still problematic as we cannot explicitly state which action rules should fire and which should not. Thus, when applied to the CHR operational semantics, all rules fire exhaustively and non-deterministically. A further problem is that the above CHRs are potentially non-terminating.

Our solution is to introduce explicit *action constraints*. Here is our actual encoding of the block world.

$$\begin{array}{rcl} get & @ & get(x), Empty, Clear(x), On(x,y) \Longleftrightarrow Holds(x), Clear(y) \\ putOn & @ & putOn(x,y), Holds(x), Clear(y) \Longleftrightarrow Empty, On(x,y), Clear(x) \end{array}$$

Each action rules is appended with an action constraint at the left-hand-side of the rule. We distinguish fluents and actions constraints by upper case and lower case terms respectively. Here is a CHR derivation after executing the actions $get(B1)$ followed by $putOn(B1, T3)$.

$$\begin{array}{ll} & \{get(B1), putOn(B1,T3), Empty, On(B1,B2), On(B2,B3), On(B3,T1), \\ & Clear(B1), Clear(T2), Clear(T3)\} \\ \rightarrowtail_{get} & \{putOn(B1,T3), Holds(B1), On(B2,B3), On(B3,T1), Clear(B2), \\ & Clear(T2), Clear(T3)\} \\ \rightarrowtail_{putOn} & \{Empty, On(B1,T3), On(B2,B3), On(B3,T1), Clear(B1), Clear(B2), \\ & Clear(T2)\} \end{array}$$

With explicit action constraints we avoid the infinite firing of rules. Though, there are still several technical issues that must be addressed:

Action Sequencing. The sequence of execution of actions according to some plan must be respected. Our solution is to introduce a monadic action CHR semantics which can neatly be explained in terms of monads [Wad95]. The CHR solver is a state monad, driven by monadic action operations which describe sequential computations of actions. Here is a possible Haskell implementation of an agent executing the action sequence $[a_1(\bar{t}_1), ..., a_n(\bar{t}_n)]$.

Agent Program	Encapsulated CHR Derivations
do	
initialize C_1	
$a_1^M(\bar{t}_1)$	$(\{a_1(\bar{t}_1)\} \uplus C_1) \rightarrowtail^*_{P_{a_1}} C_2$
$a_2^M(\bar{t}_2)$	$(\{a_2(\bar{t}_2)\} \uplus C_2) \rightarrowtail^*_{P_{a_2}} C_3$
....	....
$a_n^M(\bar{t}_n)$	$(\{a_n(\bar{t}_n)\} \uplus C_n) \rightarrowtail^*_{P_{a_n}} C_{n+1}$
end	

We assume that *initialize* C_1 initializes the CHR state monad to the initial store C_1, and $a_i^M(\bar{t}_i)$ are monadic operations associated to the action $a_i(\bar{t}_i)$. We assume that P_a are the CHRs belonging to action a. Each operation a^M has the effect of exhaustively fire the rules P_a on the current constraint store (as indicated by the CHR derivations on the right-hand side). There is clearly a design space to be explored; given that we have the action sequence $[a \mid A]$ to be executed from C and suppose that no rules in P_a can fire from C, then we must decide whether a should be 'skipped' and A executed, or computation be 'blocked' at a. The exact details of our monadic action CHR semantics are given in Section 3.2.

Action Determinism. A deterministic action is one which produces a unique effect (output) from each current state (input). Given that an action is deterministic by definition, we wish to find a mechnical way to decide if our agent CHR programs correctly represent such deterministic actions. In our approach, we use a well established CHR property, Confluence, which intuitively describes determinism of CHR programs and has a decidable test [Abd97] that can be efficiently implemented. Yet confluence is too restrictive for agent programs as it considers all states, including states that are inconsistent with our agent domains. However, by simply providing standard confluence checking routines with a set of domain invariants, inconsistent states can be accurately eliminated from consideration. In Section 4 we present the details of our approach.

3 CHRs as a Agent Specification Language

In this section we give a formal introduction to the CHR semantics. Note that we only consider CHR simplification rules. Following this, we define the monadic action CHR semantics.

3.1 CHR Syntax and Semantics

In this paper, we explore implementation of agent systems under a variant of the *theoretical operational semantics* w_t of CHR [DSdlBH04] that only allows the use of simplification rules. We shall call this variant w_s. We begin by defining some of the basic notations we will use throughout the paper. $\bar{a}$ denotes an arbitral sequence of a's. We also use $[a_1, ..., a_n]$ to denote sequences if the arity n must be explicit. We at times use the notation $[H \mid T]$ also to denote a sequence with the first element being H and the rest of the sequence T. The empty sequence is denoted as ϵ. Similar to standard notations of multisets, $\uplus$ for multiset union and $\emptyset$ for the empty set. For simplicity, we will sometimes treat

multisets as sequences, only difference being that ordering of objects are choosen non-deterministically. Finally, we define a function $fv(S) = \bar{x}$ that takes a set S and returns $\bar{x}$,the set of all occurring free variables in S.

The following defines the syntax of terms, constraints of CHR:

Term	t	::=	x	$f(\bar{t})$
Builtin Constraint	B	::=	$b(\bar{t})$	$B \wedge B$
CHR Constraint Set	C	::=	$\{u(\bar{t})\}$	$C \uplus C$

A *term* t is a variable x (we may sometimes use u, v, w, y or z, but reserve i and n for integers) or a function $f(t_1, ..., t_n)$ where arity $n \geq 0$. We define 2 types of constraints, *builtin* and *CHR constraints*, in which the former is handled by the underlying constraint solver while the latter by the CHR program. In order not to lose generality, we model *builtin constraints* as a conjunction of logical constraints $b(t_1, ..., t_n)$, where b is an abstract logical operator of arity n. We assume that the underlying solver handles at least logical equality ($t_1 = t_2$) and the *true* and *false* predicates with their obvious meanings. The *constraint theory* CT represent this abstract logical system of the solver and we write $CT \models H \supset H'$ to denote that under the constraint theory CT, H entails H', where H and H' are builtin constraints. *CHR constraints* are defined by $u(t_1, ..., t_n)$, where u is a predicate symbol. We use CHR constraints to represent fluents and action constraints. We will sometimes represent a single CHR constraint with the symbol c. A multiset of CHR constraints is either a singleton set c or a multiset union $C_1 \uplus C_2$. We write $\{c_1, ..., c_n\}$ for short of $\{c_1\} \uplus ... \uplus \{c_n\}$.

Definition 1. (Execution State) *An execution state is a tuple of the form* $\langle G, S, B\rangle$ *where goal* G *and store* S *are multisets of CHR constraints and* B *is a conjunction of builtin constraints. We use the symbol* σ *to represent an execution state.*

A reader familiar with CHR semantics would realize that our definition of execution state differs from the original in 2 main ways: removal of the *propagation history* T and definition of the constraint store S as a multiset rather than a set of *numbered constraints*. These differences are the results of removing propagation rules. The goal G, contains all constraints to be executed, while constraint store S contains all constraints that can be matched with constraint handling rules. An initial state is defined as follows.

Definition 2. (Initial State) *Given a goal* G*, a multiset of CHR constraints, the initial state of* G *is* $\langle G, \emptyset, true\rangle$.

Constraint handling rules describe multiset term rewriting among constraint sets. In w_s, we consider only rules of one form, known as *simplification rules*:

$$r \ @ \ h \iff g \mid b$$

where r is the rule name, h is a multiset of CHR constraints known as the *head*, g is a conjunction of builtin constraints known as the *guard* and b is a multiset of constraints known as the *body*. Note that we at times omit r if the rule name is not required and g if the guard is trivial (ie. $g = true$). A *CHR program* consists of a set of constraint handling rules. For convinence, we define a

function $renamed(P) = P'$ that takes a CHR program P and returns a renamed variant such that $fv(P) \cap fv(P') = \emptyset$. The operational semantics of W_s is based on the following transitions which map execution states to excution states:

$$
\begin{array}{ll}
(Solve) & \langle \{c\} \uplus G, S, B\rangle \rightarrowtail_P \langle G, S, c \wedge B\rangle \\
 & \text{where } c \text{ is a builtin constraint.} \\
(Intro) & \langle \{c\} \uplus G, S, B\rangle \rightarrowtail_P \langle G, \{c\} \uplus S, B\rangle \\
 & \text{where } c \text{ is a CHR constraint.} \\
(Apply) & \langle G, h \uplus S, B\rangle \rightarrowtail_P \langle C \uplus G, S, \theta \wedge B\rangle \\
 & \text{where } \exists (r \ @\ h' \Longleftrightarrow g \mid C) \in renamed(P) \text{ and} \\
 & \quad \exists \theta \text{ a substitution such that:} \\
 & \quad\quad \bullet\ h = \theta(h') \\
 & \quad\quad \bullet\ CT \models B \supset (\theta \wedge g)
\end{array}
$$

Definition 3. (CHR Derivations) *A CHR derivation of program* P*, denoted* $\rightarrowtail_P^*$*, represent a sequence of execution states connected by* w_s *transitions that leads to a final state* σ_n *where no* w_s *transitions are applicable. Given states* $\sigma = \langle G, S, B\rangle$ *and* $\sigma' = \langle G', S', B'\rangle$*, we write the derivation from initial state* σ *to final state* σ' *as* $\sigma \rightarrowtail_P^* \sigma'$*. For brevity, we sometimes present derivations with builtin constraints removed.*

For convinence, we define the function $State(\langle G, S, B\rangle) = C$ where $C = \theta(G \uplus S)$ such that θ is the most general unifier of G and S, and $CT \models B \supset \theta$. The multiset C represents an abstraction of $\sigma = \langle G, S, B\rangle$. We shall refer to states σ and abstract states C interchangeably. We also define derivations for abstract states.

Definition 4. (State Abstracted Derivations) *Given a multiset of CHR constraints* C*, we define the state abstracted derivation of* C *as* $C \rightarrowtail_P^* C'$ *where* C' *is a multiset of constraints, and there exists a CHR derivation* $\langle C, \emptyset, true\rangle \rightarrowtail_P^* \sigma'$ *such that* $State(\sigma') = C'$*.*

We use state abstracted derivations when we are more interested in the *apply* steps of CHR derivations, also when *solve* and *intro* steps can be abstracted.

3.2 Monadic Action CHR Semantics

In this section, we introduce the various agent oriented refinements on the CHR semantics w_s. We introduce a special type of CHR constraints known as *action constraints* which act as the explicit representation of actions. For simplicity, we restrict action rules to the following form,

$$r \ @\ a, h \Longleftrightarrow g \mid b$$

where r is the name of the action, a is an action constraint, h (qualification conditions) is a multiset of CHR constraints, g (guard) is a conjunction of builtin constraint and b (body) is a multiset of constraints. Note that h and b strictly contain no action constraints. We call rules of this form *simplification action rules* or simply action rules.

Definition 5. (CHR Agent Program) *A CHR agent program* P *is a set of simplification action rules. We write* P_a*, such that* $P_a \subseteq P$*, to denote the set*

of all action rules $(r@a, h \iff g \mid b) \in P$ *for some* r, h, g *and* b*. For short, we sometimes display CHR programs as sets of rule names.*

Lemma 1. (Termination) *Given a CHR agent program* P*, a set of action rules,* P *is terminating.*

Proof sketch: This proof depends on the restriction that action constraints cannot be found in the body of action rules and the head of actions must contain exactly an action constraint. Thus, the firing of action rules involve the 'consumption' of an action constraint. Since there can only be finitely many action constraints, P is always terminating. □

Given an execution state $\sigma = \langle G, S, B \rangle$, an action a is said to be *active* in σ if $\exists a(\bar{t}) \in S$ or $\exists a(\bar{t}) \in G$. If this active action a has an associated rule variant $(r \ @ \ a(\bar{t}), h \iff g \mid b) \in renamed(P)$ such that r may eventually fire in σ (ie. $(\{a(\bar{t})\} \uplus h) \subseteq (S \uplus G)$ and $CT \models B \supset g$), then action a is *qualified* in σ. Otherwise, it is *unqualified* in σ. By introducing action constraints, block world CHR representation P^{Bw} allows explicit control of actions in w_s operational semantics, as only rules associated to active actions are allowed to fire.

However, we still cannot represent *sequences* of actions explicitly since for an execution state $\langle G, S, B \rangle$, action constraints in the goal G are introduced to S in a non-deterministic order. We introduce the *Monadic action CHR semantics*, which describes sequential computations of agent actions. Action Monads are structures that governs the order in which actions constraints are introduced to the underlying CHR solver. Rules associated to an active action must also be allowed to fire before new actions are activated, hence guaranteeing that at most one action constraint can exist in the constraint store at every CHR derivation step. We define *action execution state* as follows:

Definition 6. (Action Execution State) *An action execution state is a tuple of the form* $(A \mid C)$ *where* A *is a sequence of actions (also known as the* plan*) and* C *is a multiset of CHR constraints that contains no action constraints.*

We view the action execution states as a monad that describes the sequential computation of actions by encapsulating the CHR derivations of individual actions within each action monadic operation. Note that in this paper, we focus entirely on the domain representation problem of agent systems and assume that our agents possess 'black box' planning routines, possibly encapsulated within monadic operations as well. We define the *Monadic action CHR derivation* $\longrightarrow_P$ by the three following transitions that connects action execution states to action execution states.

$$\frac{C \rightarrowtail_P^* C'}{(\epsilon \mid C) \longrightarrow_P (\epsilon \mid C')} \qquad \text{(Empty Sequence)}$$

$$\frac{(\{a\} \uplus C) \rightarrowtail_P^* C' \quad a \notin C'}{([a \mid A] \mid C) \longrightarrow_P (A' \mid C')} \qquad \text{(Qualified Action)}$$

$$\frac{(\{a\} \uplus C) \rightarrowtail_P^* (\{a\} \uplus C')}{([a \mid A] \mid C) \longrightarrow_P ([a \mid A] \mid C')} \qquad \text{(Unqualified Action)}$$

The action monads represent the clear dividing line between the sequential computations of agent actions and the commited choice CHR derivations

describing effects of each individual actions, allowing the two unidentical semantics to coexist and function supportively. The first transition (Empty Sequence) describes the base case, where the action sequence is empty. The next two transitions describe cases where action a is active in abstract state C. For the former is the case where a qualifies in C and the exhaustive CHR derivation from the state $(\{a\} \uplus C)$ must result in the firing of a rule $r \in P_a$. The latter is the case where action a is unqualified in C, in which further computation of the action sequence A is not possible as computation *blocks* at the action a.

The choice of the *blocking semantics* of unqualified actions is a design decision that has been made with much considerations. No doubt for $[a \mid A]$ we could have modeled unqualified action a to be 'skipped' and have the rest of the actions A executed as usual, yet this semantics is often undesirable: Given an action execution state $\langle [a \mid A] \mid C \rangle$, the fact that action a is unqualified in C implies that either the agent's planning routines has constructed an inapplicable plan because of unforseen changes in the agent domain, or simply that the desirable qualification conditions of the action a has not been realised yet and the action should be attempted again at a later stage. Either way, the most intuitive response of the action monad is to return the last known action execution state that was successfully computed, from which the agent's planning routines can further deliberate appropriate responses. This corresponds exactly to the blocking semantics describe by the unqualified action transition above.

The monadic action CHR semantics guarantees an important property which we denote *action isolation*. Action isolation states implies that our agent CHR solver is never put in a situation where more than one action constraint is in the constraint store.

Lemma 2. (Action Isolation) *Given an action sequence A_0 and an execution state C_0 such that C_0 contains no action constraints, for all action derivation sequences $(A_0 \mid C_0) \longrightarrow_P .... \longrightarrow_P (A_n \mid C_n)$, all C_i contains no action constraints.*

The proof sketch of lemma 2 is as follows: We proof by induction on action execution states. Assume that C_0 contains no action constraints and CHR derivation $C_0 \rightarrowtail_P^* C_1$, for the empty sequence case, we know the state C_1 that contains no action constraints, since all rules in P never introduce new action constraints. For the qualified action case, suppose the active action is a, an action rule associated to a must have fired and consumed a, thus C_1 contains no action constraints. For unqualified action case, C_1 indeed contains exactly one action constraint a, (ie. $C_1 = \{a\} \uplus C_2$) . However, output of the unqualified action transition is C_2 which contains no action constraint. □

Note that action isolation guarantees that any monadic action derivation $([a_1, ..., a_n] \mid C_1) \longrightarrow_P^* (\epsilon \mid C_{n+1})$ effectively produces the CHR derivation sequence $(\{a_1\} \uplus C_1) \rightarrowtail_{P_{a_1}}^* C_2$,..., $(\{a_n\} \uplus C_n) \rightarrowtail_{P_{a_n}}^* C_{n+1}$.

4 Action Determinism

In this section we introduce a framework to reason about properties of CHR agent systems. In particular, we examine *determinism of actions* defined by the following.

Definition 7. (Action Determinism) *An action a of an agent domain is* Deterministic *iff executing the action a from any agent domain state results in a unique resultant state.*

Both block world actions (*get* and *putOn*) are deterministic according to the block world specifications, thus any formal specification or implementation of these actions must correctly reflect this property. We introduce a practical way to verify the determinism of agent actions via CHR *confluence*. For the purpose of capturing cases of actions with multiple rules, we consider an extension of the block world, denoted *block world prime* $P^{Bw'}$.

$$\begin{array}{rcl} get1 & @ & get(x), Empty, Clear(x), On(x,y) \iff Holds(x), Clear(y) \\ get2 & @ & get(x), Holds(z), Clear(x), On(x,y) \iff Holds(x), Clear(y), Thrown(z) \\ putOn & @ & putOn(x,y), Holds(x), Clear(y) \iff Empty, Clear(x), On(x,y) \end{array}$$

Block world prime extends block world by allowing the *get* action to be executed even if the robotic arm is holding some other object z. The side effect is that z will be thrown away, denoted by the fluent $Thrown(z)$.

4.1 Confluence

We now define *state variants*, *joinability* and *confluence* under the theoretical semantics w_s.

Definition 8. (Variants) *Two execution states* $\sigma = \langle G, S, B\rangle$ *and* $\sigma' = \langle G', S', B'\rangle$ *are said to be variants* ($\sigma \approx \sigma'$) *if there exists a variable rename* ρ*, such that* $\rho(G) = G'$, $\rho(S) = S'$ *and* $CT \models B \leftrightarrow B'$.

Definition 9. (Joinable) *Given a CHR program P and two execution states,* σ_1 *and* σ_2*, are joinable if there exists 2 states* σ_1' *and* σ_2' *such that* $\sigma_1 \rightarrowtail^* \sigma_1'$ *and* $\sigma_2 \rightarrowtail^* \sigma_2'$ *and* σ_1' *and* σ_2' *are variants.*

Definition 10. (Confluence) *A CHR program P is confluent iff given the states* σ_0, σ_1 *and* σ_2*, we have the following: if* $\sigma_0 \rightarrowtail_P^* \sigma_1$ *and* $\sigma_0 \rightarrowtail_P^* \sigma_2$ *then* σ_1 *and* σ_2 *are joinable.*

Confluence provides a more suitable notion of determinism for CHR programs. In [Abd97], an efficient way of testing for confluence is introduced. This *confluence test* is based on the testing of all *critical pairs* between each rules of the CHR program. We define critical pairs and confluence test with respect to the w_s semantics.

Definition 11. (Critical Pairs & Origins) *Given a CHR program P, for any 2 rules* $(r_1 \ @ \ H_1 \iff g_1 \mid B_1)$ *and* $(r_2 \ @ \ H_2 \iff g_2 \mid B_2)$ *such that* $r_1, r_2 \in P$. *Let* $H_1' \subseteq H_1$ *and* $H_2' \subseteq H_2$ *and* θ *be the most general unifier of* H_1' *and* H_2'. *Also define* S, S_1 *and* S_2 *such that* $S = \theta(H_1) \uplus (H_2 - H_2')$, $S_1 \subseteq S$ *and* $S_1 = \theta(H_1)$, *and* $S_2 \subseteq S$ *and* $S_2 = \theta(H_2)$, *then the states*

$$\sigma_1 = \langle B_1, S - S_1, g_1 \wedge g_2\rangle \qquad \sigma_2 = \langle B_2, S - S_2, g_1 \wedge g_2\rangle$$

forms a critical pair (σ_1, σ_2) *between rules* $r1$ *and* $r2$. *We say that the critical pair* σ_1 *and* σ_2 *originates from* S.

Definition 12. (Confluence Test) *Given a terminating CHR program P, the confluence test on P is as follows: if all critical pairs of P are joinable, then P is confluent.*

Note that the confluence test is applicable only for terminating CHR programs. However, this will not be a problem, since we are interested only in terminating CHR agent programs. It would seems confluence test is perhaps a candidate for testing determinism of actions, yet $P^{Bw'}$ is not confluent even though it is obviously deterministic. Consider the state $\sigma_0 = \langle\{\}, \{get(x), Empty, Holds(w), Clear(x), On(x,y), On(x,w)\}, true\rangle$. The following shows derivations resulting to a non-joinable critical pair (σ_1, σ_2), between the rules $get1$ and $get2$.

$$\begin{array}{l} \langle\{\}, \{get(x), Empty, Holds(w), Clear(x), On(x,y), On(x,w)\}, true\rangle \\ \rightarrowtail_{get2} \langle\{Holds(x), Clear(z), Thrown(w)\}, \{Empty, On(x,y)\}, true\rangle \end{array}$$

$$\begin{array}{l} \langle\{\}, \{get(x), Empty, Holds(w), Clear(x), On(x,y), On(x,w)\}, true\rangle \\ \rightarrowtail_{get1} \langle\{Holds(x), Clear(y)\}, \{Holds(w), On(x,w)\}, true\rangle \end{array}$$

On closer observation, we notice that the origin (σ_0) of the critical pairs σ_1 and σ_2, describes a state that is inconsistent with block world domain invariant $\mathcal{I}_1^{Bw'}$ and should never be reachable (Robotic arm cannot be both $Empty$ and $Holds(w)$). In fact, all other non-joinable critical pairs of $P^{Bw'}$ either originates from states that do not conform to the block world specifications or are critical pairs between rules of different actions (action isolation of monadic action CHR derivation guarantees these never occur). Due to space constraints, we will not enumerate all such critical pairs here, but invite the reader to replicate this observation.

Confluence test is too restrictive in testing for action determinism because it considers all critical pairs, including those which may not be reachable from any initial state. What we need is a more refined confluence test that tests only *reachable* critical pairs for joinability and ignores unreachable ones.

Definition 13. (Reachability) *Given a CHR program P, a state C is reachable iff either C is an initial state or there exists a CHR derivation $C' \rightarrowtail_P^* C$ for some initial state C'.*

We define a weaker notion of confluence, called $\mathcal{I}$-confluence and the $\mathcal{I}$-confluence test as follows:

Definition 14. ($\mathcal{I}$-Confluence) *A CHR program P is confluent iff given the states σ_0, σ_1 and σ_2 such that σ_0 is reachable, we have the following: if $\sigma_0 \rightarrowtail_P^* \sigma_1$ and $\sigma_0 \rightarrowtail_P^* \sigma_2$ then σ_1 and σ_2 are joinable.*

Definition 15. ($\mathcal{I}$-Confluence Test) *Given a terminating CHR program P the $\mathcal{I}$-confluence test on P is as follows: For all critical pairs (σ_1, σ_2) of P, if origin of (σ_1, σ_2) is reachable and σ_1 and σ_2 are joinable, then P is $\mathcal{I}$-confluent.*

The $\mathcal{I}$-confluence of a program P immediately implies the determinism of all action rules of P. However, we still need a more concrete test for reachable state. For this, we look to domain invariants.

4.2 Domain Invariants

Domain invariants of block world prime ($\mathcal{I}^{Bw'}_{1-5}$) were briefly mentioned informally in section 2 (We assume that $\mathcal{I}^{Bw'}_{1-5} = \mathcal{I}^{Bw}_{1-5}$). These domain invariants can be generalized to the 2 following forms.

Definition 16. (Mutual Exclusion) *Given two CHR constraints $P(\bar{x})$ and $Q(\bar{y})$, and an state C, $P(\bar{x})$ and $Q(\bar{y})$ are mutually exclusive in C iff*

$$\forall \bar{z}.(P(\bar{x}) \in C) \supset (Q(\bar{y}) \notin C) \wedge (Q(\bar{y}) \in C) \supset (P(\bar{x}) \notin C) \text{ where } \bar{z} = fv(\bar{x}) \cup fv(\bar{y})$$

For brevity, we denote mutual exclusion of p_1 and p_2 in C as $\forall \bar{t}\ MX(p_1, p_2, C)$ where $\bar{t} = fv(p_1) \cup fv(p_2)$.

Definition 17. (Functional Dependency) *Given a CHR constraint $P(x_1, .., x_n)$, and an state C, the functional dependency $\{x_{i_0}, .., x_{i_j}\} \rightsquigarrow \{x_1, .., x_n\}$ where $\{x_{i_0}, .., x_{i_j}\} \subseteq \{x_1, .., x_n\}$ exists in C, iff*

$$\begin{array}{l} \forall x_1, .., x_n, y_1, .., y_n. \\ \quad (P(x_1, .., x_n) \in C \wedge P(y_1, .., y_n) \in C \wedge x_{i_0} = y_{i_0} \wedge .. \wedge x_{i_j} = y_{i_j}) \\ \qquad \supset (x_1 = y_1 \wedge .. \wedge x_n = y_n) \end{array}$$

For brevity, we denote function dependency, $\bar{x}' \rightsquigarrow \bar{x}$ where $\bar{x}' \subseteq \bar{x}$, of $P(x_1, .., x_n)$ in S as $\forall \bar{x}\ FD(P(\bar{x}), \bar{x}', C)$.

A third form of domain invariant *uniqueness* is required for a more technical reason that will be clear soon. Uniqueness of a CHR constraint $P(\bar{t})$ in C dictates that we can only have exactly one copy of $P(\bar{t})$ in the multiset C.

Definition 18. (Uniqueness) *Given a CHR constraint $P(\bar{x})$, and an state C, $P(\bar{x})$ is unique in C iff*

$$\begin{array}{l} \forall \bar{x}, n, m.(C' = label(C) \wedge (P(\bar{x})\#n) \in C' \wedge (P(\bar{x})\#m) \in C') \supset m = n \\ \text{where } label(\{c_1, .., c_i, .., c_n\}) = \{(c_1\#1), .., (c_i\#i), .., (c_n\#n)\} \end{array}$$

For brevity, we denote uniqueness of $P(\bar{x})$ in C as $\forall \bar{x}\ UN(P(\bar{x}), C)$.

We formally define domain invariants $\mathcal{I}$ as a conjunction of first order logic formulas of the above forms. We write $\mathcal{I}(C)$ to denote that the state C satisfies the domain invariant $\mathcal{I}$. We define block world initial states as follows.

Definition 19. (Block world Initial States) *Given a state C, C is a block world initial state if $\mathcal{I}(C)$.*

Note that we omit the obvious that initial state C must contain only block world fluents. An important property of the invariants $\mathcal{I}$ is as follows:

Lemma 3. (Invariance of Subsets) *Given a state C and invariants $\mathcal{I}$, if $\mathcal{I}(C)$ and $C' \subseteq C$, then $\mathcal{I}(C')$*

We omit a formal proof of lemma 3 because it follows almost immediately from the definition of the 3 forms of invariants. Given invariants $\mathcal{I}$, we are interested if a CHR program *preserves* $\mathcal{I}$. We define this by the following

Definition 20. (Preserving Invariants) *Given the invariants $\mathcal{I}$ and a CHR program P, P preserves $\mathcal{I}$ iff for any state C such that $\mathcal{I}(C)$ and we have the derivation $C \rightarrowtail_P C'$ for some state C', then $\mathcal{I}(C')$.*

We can now formally define the block world domain invariants $\mathcal{I}_{1-5}^{Bw'}$. The following shows that formal representation of the domain invariants:

$$
\begin{aligned}
\mathcal{I}_1^{Bw'}(C) &= \forall x.\ MX(Empty, Holds(x), C)\\
\mathcal{I}_2^{Bw'}(C) &= \forall x, y.\ MX(On(x,y), Clear(y), C)\\
\mathcal{I}_3^{Bw'}(C) &= \forall x, y.\ FD(On(x,y), \{x\}, C) \wedge \forall x, y.\ FD(On(x,y), \{y\}, C)\\
\mathcal{I}_4^{Bw'}(C) &= \forall x.\ FD(Holds(x), \emptyset, C)\\
\mathcal{I}_5^{Bw'}(C) &= \forall x, y.\ MX(Holds(x), On(x,y), C) \wedge \forall x, y.\ MX(Holds(y), On(x,y), C) \wedge\\
&\quad\ \forall x.\ MX(Holds(x), Clear(x), C)
\end{aligned}
$$

We introduce the following axilliary invariants:

$$
\begin{aligned}
\mathcal{I}_U^{Bw'}(C) &= \forall x.\ UN(Holds(x), C) \wedge \forall x, y.\ UN(On(x,y), C) \wedge\\
&\quad\ \forall x.\ UN(Clear(x), C) \wedge UN(Empty, C)\\
\mathcal{I}_A^{Bw'}(C) &= \forall x, y.\ MX(get(x), get(y), C) \wedge \forall x, y, z.\ MX(get(x), putOn(y,z), C) \wedge\\
&\quad\ \forall x, y, w, z.\ MX(putOn(x,y), putOn(w,z), C)
\end{aligned}
$$

Things can go wrong if a block world state C is allowed to have multiple copies of certain fluents. Thus $\mathcal{I}_U^{Bw'}$ ensures all fluents of block world must be unique (Note that in general, we allow multisets). Also, recall the *action isolation* (section 3.2) property that the monadic action CHR derivations guarantee. We express this with $\mathcal{I}_A^{Bw'}$, which states mutual exclusion of all action constraints. The complete block world invariants are represented by:

$$\mathcal{I}^{Bw}(C) = \mathcal{I}_1^{Bw'}(C) \wedge \mathcal{I}_2^{Bw'}(C) \wedge \mathcal{I}_3^{Bw'}(C) \wedge \mathcal{I}_4^{Bw'}(C) \wedge \mathcal{I}_5^{Bw'}(C) \wedge \mathcal{I}_U^{Bw'}(C) \wedge \mathcal{I}_A^{Bw'}(C)$$

Lemma 4. ($P^{Bw'}$ Invariance) *Block world program $P^{Bw'}$ preserves $\mathcal{I}^{Bw'}$*

The proof of lemma 4 involves proving that each rule $r \in P^{Bw'}$ preserves each atomic invariant $\mathcal{I}_i^{Bw'}$ of $\mathcal{I}^{Bw'}$. We shall sketch the proof for $get1$, which will provide the general recipe of the proof. given a state C such that $\mathcal{I}^{Bw'}(C)$, by lemma 3, we can infer that removing constraints from C will not violate $\mathcal{I}^{Bw'}$, thus for $get1$ rule, we only need to check if invariants containing references of $Holds(x)$ and $Clear(y)$ is violated. Since we consider the derivation where $get1$ has fired, we must have $C \rightarrowtail_{get1} C'$ such that $C = \{get(X), Empty, Clear(X), On(X,Y)\} \uplus C''$ and $C' = \{Holds(X), Clear(Y)\} \uplus C''$ for some C'' and skolem constants X and Y. Now we prove $\mathcal{I}^{Bw'}(C')$. The enumeration of the proof is as follows, recalling for each we have $\mathcal{I}^{Bw'}(C)$:

- Prove $\mathcal{I}_1^{Bw'}(C')$. Since $C = \{get(X), Empty,\ Clear(X), On(X,Y)\} \uplus C''$ and $UN(Empty, C)$, then $Empty \notin C''$. Thus, $Empty \notin C'$, implying $\mathcal{I}_1^{Bw'}(C')$.

- Prove $\mathcal{I}_2^{Bw'}(C')$. Similar to above, we have $On(X,Y) \notin C''$, Thus $On(X,Y) \notin C'$ and $\mathcal{I}_2^{Bw'}(C')$.
- Prove $\mathcal{I}_4^{Bw'}(C')$. We need $\forall z.\ Holds(z) \notin C''$. Since $Empty \in C$, $\mathcal{I}_1^{Bw'}(C)$ dictates that $\forall z.\ Holds(z) \notin C''$.
- Prove $\mathcal{I}_5^{Bw'}(C')$. First we need $\forall z.\ MX(Holds(X), On(X,z), C')$ which we get from $On(X,Y) \notin C''$ and $\mathcal{I}_3^{Bw'}(C)$. Next, we get $\forall z.\ MX(Holds(X), On(z,X), C')$ from $Clear(X) \in C$ and $\forall z.\ MX(On(z,X), Clear(X), C)$. Finally $MX(Holds(X), Clear(X), C')$ is simply obtained from $Clear(X) \notin C''$.
- Prove $\mathcal{I}_U^{Bw'}(C')$. We need $UN(Holds(X), C')$ and $UN(Clear(Y), C')$. Since $Empty \in C$, $\forall z.Holds(z) \notin C''$, by $\mathcal{I}_1^{Bw'}(C)$, hence $UN(Holds(X), C')$. Since $On(X,Y) \in C$ $Clear(Y) \notin C''$ as $\mathcal{I}_2^{Bw'}(C)$. Therefore $UN(Clear(Y), C')$.

Thus, we have $\mathcal{I}^{Bw'}(C')$ for the $get1$ rule. Notice that the uniqueness of fluents of block world are necessary for the proof. The proof for the $get2$ and $putOn$ rules can be easily replicated from the above recipe. Note that the reasoning involved in the proof above is rather simple and mechanical, suggesting possibility for efficient implementations if domain invariants are well represented. □

Lemma 5. (Block world Soundness) *With respect to program $P^{Bw'}$, given a state C, if C is reachable, then $\mathcal{I}^{Bw'}(C)$.*

Proof Sketch of lemma 5 is as follows: there are 2 cases of C. First, if C is a block world initial state, then we get this by definition 4.2. Otherwise we have $C' \rightarrow^* C$ for some initial state C, which means $\mathcal{I}^{Bw'}(C')$ By lemma 4, we get $\mathcal{I}^{Bw'}(C)$. □

4.3 $\mathcal{I}$-Confluence of Block World

With the domain invariants of block world $\mathcal{I}^{Bw'}$ and the soundness of block world with respect to the invariants, we can prove the $\mathcal{I}$-confluence of block world.

Lemma 6. ($\mathcal{I}$-Confluence of Block World) *$P^{Bw'}$ is $\mathcal{I}$-confluent under $\mathcal{I}^{Bw'}$.*

Proof Sketch of lemma 6: We begin by setting up the use of the invariants $\mathcal{I}^{Bw'}$ to identify reachable states of block world. We use lemma 5 which states that C is reachable implies that $\mathcal{I}^{Bw'}(C)$ Given a critical pair (σ_1, σ_2) that originates from a state S, we test reachability of (σ_1, σ_2) by testing $\mathcal{I}^{Bw'}(S)$. If $\mathcal{I}^{Bw'}(S)$ then we test if σ_1 and σ_2 are joinable. Otherwise (σ_1, σ_2) is unreachable: Domain invariants $\mathcal{I}_{1-5}^{Bw'}$ prunes away states that are inconsistent with block world, while $\mathcal{I}_U^{BW'}$ prunes away states with duplicate block world fluents and $\mathcal{I}_A^{Bw'}$ prunes away states violating action isolation (lemma 2). The second part of the proof involves the enumeration of all critical pairs of $P^{Bw'}$ and pruning away critical pairs that are unreachable, and testing all others for joinability. We omit this part of the proof but insist that it is tedious but straight forward. □

Note that this testing method is in fact only an approximate for $\mathcal{I}$-confluence test: Passing the test guarantees $\mathcal{I}$-confluence, but failing does not guarantee non $\mathcal{I}$-confluence. In general, the set of all states S such that $\mathcal{I}(S)$ is a super set of the 'ideal' set of all reachable states of the CHR program.

5 Conclusion

We have laid out the foundations for an approach towards programming of linear logic agents via CHRs. Linear logics provide a strong and sound logical foundation for the formal description and reasoning of dynamic agent systems, while CHRs provide an executable framework for testing and running simulations. We have investigated the necessary refinements of the standard CHR semantics such as action constraints and the monadic action CHR semantics.

Our approach in using monads to combine computation paradigms shares similarities with the monadic concurrent linear logic programming proposed in [LPPW05] called LolliMon. LolliMon has a natural forward chaining, committed choice operational semantics inside the monad, and an asynchronous linear logic based backward chaining opertional semantics outside the monad. The $\mathcal{I}$-confluence we have defined for determinism test is closely related to observable confluence explored in [DSS06]. These resemblances are yet to be studied in detail.

In future work, we plan to develop a general $\mathcal{I}$-confluence test for CHR programs with reachable states approximated by program annotations denoting domain invariants. We also intend to extend the framework by introducing propagation rules to the framework.

References

[Abd97] S. Abdennadher. Operational Semantics and Confluence of Constraint Propagation Rules. In *Proc. of CP'97*, pages 252–266, 1997.

[BF05] H. Betz and T. Frühwirth. A Linear-Logic Semantics for Constraint Handling Rules. In *Proc of CP'2005*, volume 3709 of *LNCS*, pages 137–151. Springer-Verlag, 2005.

[BMR95] A. Borgida, J. Mylopoulos, and R. Reiter. On the Frame Problem in Procedure Specifications. *IEEE Trans. Software Eng.*, 21(10):785–798, 1995.

[DSdlBH04] G. J. Duck, P. J. Stuckey, M. J. García de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *Proc of ICLP'04*, pages 90–104, 2004.

[DSS06] G. J. Duck, P. J. Stuckey, and M. Sulzmann. Observable Confluence for Constraint Handling Rules, 2006. In this volume.

[Frü95] T. Frühwirth. Constraint Handling Rules. *Lecture Notes in Computer Science*, (910):90–107, 1995.

[Gir95] J. Y. Girard. Linear Logic: Its Syntax and Semantics. In *Proc. of Workshop on Linear Logic, Cornell University*, number 222. Cambridge University Press, 1995.

[LPPW05] P. López, F. Pfenning, J. Polakow, and K. Watkins. Monadic Concurrent Linear Logic Programming. In *Proc. of PPDP*, pages 35–46, 2005.

[MH69] J. McCarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[MNPW98] N. Muscettola, P. Pandurang Nayak, B. Pell, and B. C. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[Thi98] M. Thielscher. Introduction to the Fluent Calculus. *Electron. Trans. Artif. Intell.*, 2:179–192, 1998.

[Wad95] P. Wadler. Monads for Functional Programming. In *Advanced Functional Programming*, pages 24–52, 1995.

Implementation of an F-Logic Kernel in CHR

Martin Kaeser and Marc Meister

Universität Ulm, Germany
{Martin.Kaeser,Marc.Meister}@uni-ulm.de

Abstract. Constraint Handling Rules (CHR) is a concurrent, committed-choice, rule-based language, rewriting constraints in relational syntax. Frame-Logic is an extension of classical predicate logic which accounts in a declarative way for many features of object-orientation. This exploratory paper gives a concise CHR implementation of Frame-Logic's core features, including object-oriented constraint syntax, type-checking, and interaction of Frame-Logic deduction with non-monotonic overriding by inheritance.

1 Introduction

Constraint Handling Rules (CHR) [4] is a concurrent, committed-choice, rule-based language which was originally developed for rapid prototyping and for writing constraint solvers.

Frame-Logic (F-Logic) [6] is an extension of classical predicate logic, which accounts in a declarative way for many features of object-orientation: Object-oriented syntax breaks the so-called *impedance mismatch* between (relational) logic and object-oriented programming. Object methods are encapsulated and not scattered around relations. Overriding of inherited definitions, common in standard programming languages like Java or C++, is modelled by allowing inheritance to be non-monotonic. F-Logic allows intricate interaction between rule execution, i.e., F-Logic deduction, and inheritance [11]. Our motivation for using F-Logic is that object-orientation is described by F-Logic as relational programming is by classical predicate logic. As the classical declarative reading of a CHR program is given in classical predicate logic, an object-oriented extension of CHR should be described in the object-oriented F-Logic.

F-Logic is orthogonal to other extensions of classical predicate logic like (non-monotonic) transaction logic [2] and (syntactically higher-level) HiLog [3]. All three extensions are implemented in the elaborate FLORA-2 system [12, 5], which is based on general logic programming and uses XSB's [9] implementation of the three-valued well-founded semantics. However, a thorough background both in theory and implementation is a prerequisite for using FLORA-2, so it would be much nicer to express parts of F-Logic in terms of a programmable language such as CHR. In this exploratory paper we show that our CHR implementation accounts for the basic F-Logic features including object-oriented syntax, F-Logic deduction, inheritance, and type-checking.

Beside the trend to use CHR as general purpose programming language, we see two long-term benefits of investigating CHR and object orientation (cf. [7]): First, object-oriented syntax and non-monotonic inheritance semantics is crucial to use CHR as reasoning engine in mainstream software engineering. Second, understanding object-orientation in terms of a rule-based language should make it easier to apply mainstream object-oriented software engineering principles on a to be devised object-oriented CHR dialect.

Our paper is organised as follows: After introducing F-Logic in Section 2, we describe our implementation in detail in Section 3, and conclude.

Complete sources and examples for our implementation (for SICStus Prolog [8]) are available [1].

2 F-Logic

We give an introduction to F-Logic: An example illustrates important aspects of F-Logic (Section 2.1). We describe the syntax of F-Logic expressions (Section 2.2) and explain the semantics of inference both by rules and by inheritance and their interaction (Section 2.3).

In order to exhibit the core concepts of F-Logic we focus on set-valued method definitions only (as proposed by Kifer [5]).

2.1 Motivating Example

We give an overview of F-Logic by means of an example. An F-Logic program contains *F-facts*, *is-a atoms*, and *object atoms*. The relation between objects is defined by *is-a atoms*:

$$kaylee : goodguy,\ simon : goodguy,\ jayne : badguy,\ mal : person,$$
$$goodguy :: person,\ badguy :: person.$$

$a :: b$ defines a *subclass relationship*, while $a : b$ defines a *class membership*. For example, *kaylee* is a class member (instance) of *goodguy*, which is a subclass of *person*.

An *object atom* $o[p]$ defines a *method expression* p for an object o:

$$person[attitude \bullet\!\!\rightarrow\!\!\!\!\rightarrow friendly],$$
$$badguy[attitude \bullet\!\!\rightarrow\!\!\!\!\rightarrow aggressive],$$
$$kaylee[attitude \rightarrow\!\!\!\!\rightarrow shiny].$$

The *data expression* $b \rightarrow\!\!\!\!\rightarrow c$ is non-inheritable, while $b \bullet\!\!\rightarrow\!\!\!\!\rightarrow c$ is inheritable, i.e., influencing additionally its subclasses and instances. Methods are always set-valued, so multiple expressions are possible with different values for the same object and method. For example, in addition to $kaylee[attitude \rightarrow\!\!\!\!\rightarrow shiny]$ the expression $kaylee[attitude \rightarrow\!\!\!\!\rightarrow aggressive]$ is allowed.

Objects and *classes* are not mutually exclusive, e.g., *person* has class properties (i.e., it has instances and subclasses) *and* object properties (i.e., it has method expressions), while *kaylee* has only object properties.

F-Logic provides implicit inference by inheritance. If not overwritten, inheritable expressions are passed to its subclasses also as inheritable, but to its instances as non-inheritable: *goodguy* is a subclass of *person*, so the expression $goodguy[attitude \bullet\!\!\rightarrow\!\!\!\!\rightarrow friendly]$ is inherited. The objects *simon* and *kaylee* are instances of *goodguy*, so *simon* inherits $attitude \rightarrow\!\!\!\!\rightarrow friendly$, but *kaylee* overrides *attitude* to be *shiny*.

Information is inferred explicitly by F-rules, which may be part of an F-Logic program:

$$X[likes \rightarrow\!\!\!\!\rightarrow X] \longleftarrow X : badguy,$$
$$X[likes \rightarrow\!\!\!\!\rightarrow Y] \longleftarrow X : goodguy \; and \; Y : goodguy.$$

If the expression in the body is satisfied for some variable assignment, the atom in the head is also considered true. The examples tell that every *goodguy likes* each *goodguy*, while every *badguy* only *likes* himself, so the following F-facts are inferred:

$$jayne[likes \rightarrow\!\!\!\!\rightarrow jayne],$$
$$kaylee[likes \rightarrow\!\!\!\!\rightarrow simon],\; kaylee[likes \rightarrow\!\!\!\!\rightarrow kaylee],$$
$$simon[likes \rightarrow\!\!\!\!\rightarrow simon],\; simon[likes \rightarrow\!\!\!\!\rightarrow kaylee].$$

F-Logic supports F-queries, i.e., for an expression with (possibly) variables, e.g., we ask for those variable assignments that satisfy

$$? - X[likes \rightarrow\!\!\!\!\rightarrow kaylee] \; and \; X[attitude \rightarrow\!\!\!\!\rightarrow Y].$$

and get the answer

$$(X = kaylee \; and \; Y = shiny) \; or \; (X = simon \; and \; Y = friendly)$$

2.2 Syntax

Since the original reference [6], the syntax of F-Logic was modernised and simplified. We distinguish set/single-valued F-expressions by cardinality constraints (as proposed in [5]), and use only one type of arrow: The two-headed arrow fits Prolog's syntax nicely.

Following the usual convention, variables start with uppercase letters and function symbols start with lowercase letters. An *id-term* is a first-order term composed of function symbols and variables, A *logical object id* (short: *oid*), that is used to identify an object, is a *ground* (i.e., variable-free) id-term.

F-atoms are the basic elements of F-Logic:

is-a atom Defines a relation between two id-terms: $A : B$ indicates that A is a *class member (instance)* of B, while $A :: B$ means that A is a *subclass* of B. Example: $simon : goodguy$, $goodguy :: person$, $X : object$.

object atom $O[E]$ defines a *method expression* E for an id-term O. E is either a *data* expression ($M \bullet\!\!\rightarrow\!\!\!\!\rightarrow V$, $M \rightarrow\!\!\!\!\rightarrow V$) or a *signature* expression ($M \bullet\!\!\Rightarrow\!\!\!\!\Rightarrow V$, $M \Rightarrow\!\!\!\!\Rightarrow V$), that each can be *inheritable* or *non-inheritable*.
Example: $person[likes \bullet\!\!\Rightarrow\!\!\!\!\Rightarrow person]$, $kaylee[attitude \rightarrow\!\!\!\!\rightarrow shiny]$.

F-formulas are expressions constructed of F-atoms, the usual logical connectives *negation*, $\wedge$, $\vee$, and *quantifiers*. Special F-formulas are:

F-query Expression consisting of F-atoms and the logical connectives $\wedge$ and $\vee$.
F-rule $H \longleftarrow B$ for a universally quantified expression $\forall(\neg B \vee H)$ (H is a conjunction of F-atoms, and B is an F-query).
F-fact Special case of F-rule: $F \longleftarrow true$.

An *F-molecule* is a compact notation for a conjunction of F-atoms, e.g., $a : c[m \bullet\!\!\rightarrow [b1, b2], m \bullet\!\!\Rightarrow c]$ means $a : c \wedge a[m \bullet\!\!\rightarrow b1] \wedge a[m \bullet\!\!\rightarrow b2] \wedge a[m \bullet\!\!\Rightarrow c]$.

We call F-facts, F-rules and F-queries consisting of F-molecules instead of F-atoms *molecular*.

2.3 Semantics: Inheritance and Rule Application

An *F-Logic program* (short: *F-program*) P is a conjunction of F-rules. The corresponding *Herbrand model* M is the minimal set of *ground* F-atoms, that satisfies the F-Logic program:

$$M_1 \wedge ... \wedge M_n \models P \qquad \text{for } M = \{M_1, ..., M_n\}$$

Obviously, every F-fact from the F-program is part of the model. Further, for every variable assignment, for which the body of an F-rule is satisfied by the model, the head is part of the model, e.g.:

$$\frac{P = kaylee[likes \twoheadrightarrow simon] \ \wedge \ \forall(X[likes \twoheadrightarrow Y] \longleftarrow Y[likes \twoheadrightarrow X])}{M = \{kaylee[likes \twoheadrightarrow simon], \ simon[likes \twoheadrightarrow kaylee]\}}$$

We distinguish *structural inheritance* for is-a and inheritable signature atoms, and *behavioural inheritance* for inheritable data atoms.

The model for monotonic, structural inheritance satisfies:

$$\begin{aligned} A :: B \in M \ \wedge \ B :: C \in M &\implies A :: C \in M \\ A : B \in M \ \wedge \ B :: C \in M &\implies A : C \in M \\ A :: B \in M \ \wedge \ B[C \bullet\!\!\Rightarrow D] \in M &\implies A[C \bullet\!\!\Rightarrow D] \in M \\ A : B \in M \ \wedge \ B[C \bullet\!\!\Rightarrow D] \in M &\implies A[C \Rightarrow D] \in M \end{aligned}$$

Behavioural inheritance is non-monotonic, i.e., an additional F-atom can lead to the removal of another F-atom in the model. Model-theoretic semantics is rather complicated, informally, we expect

$$\begin{aligned} A :: B \in M \ \wedge \ B[C \bullet\!\!\rightarrow D] \in M &\implies A[C \bullet\!\!\rightarrow D] \in M \\ A : B \in M \ \wedge \ B[C \bullet\!\!\rightarrow D] \in M &\implies A[C \rightarrow D] \in M \end{aligned}$$

unless the inheritance is overridden by an F-atom in an intermediate or the destination object. Consider this F-program:

$$kaylee : goodguy \ \wedge \ goodguy :: person \ \wedge \ person[attitude \bullet\!\!\rightarrow friendly]$$

Hence, $kaylee : person$ is inherited structurally, while two F-atoms are added to the model by behavioural inheritance:

$$goodguy[attitude \bullet\!\!\rightarrow\!\!\!\!\rightarrow friendly],\ kaylee[attitude \rightarrow\!\!\!\!\rightarrow friendly]$$

If $goodguy[attitude \bullet\!\!\rightarrow\!\!\!\!\rightarrow shiny]$ is added, the last two F-atoms are removed from the model. Inheritance is blocked, as for *goodguy* the destination object is overridden, while for *kaylee*, an intermediate object is overridden. Instead *kaylee* inherits from *goodguy*: $kaylee[attitude \rightarrow\!\!\!\!\rightarrow shiny]$.

The interaction of F-rules and non-monotonic inheritance leads to several subtle problems (c.f. [5]), e.g.,

$$a :: b \ \wedge\ b :: c \ \wedge\ c[m \bullet\!\!\rightarrow\!\!\!\!\rightarrow v1] \ \wedge\ b[m \bullet\!\!\rightarrow\!\!\!\!\rightarrow v2] \longleftarrow a[m \bullet\!\!\rightarrow\!\!\!\!\rightarrow v1]$$

Since no overriding occurs, $a[m \bullet\!\!\rightarrow\!\!\!\!\rightarrow v1]$ should be inherited, therefore $b[m \bullet\!\!\rightarrow\!\!\!\!\rightarrow v2]$ should be part of the model. But this method should inhibit the first inheritance in favour of $a[m \bullet\!\!\rightarrow\!\!\!\!\rightarrow v2]$, and this would undermine the F-rule application.

The original solution from Kifer [6] was not model-theoretic and was problematic in several other respects as well. A satisfactory and completely model-theoretic solution was proposed by Yang [10, 11].

3 CHR Implementation of F-Logic

The purpose of our F-Logic handler for CHR is making important parts of F-Logic executable, but keeping the implementation concise and comprehensible. To this end, our implementation covers (up to now), only a subset of the very powerful F-Logic formalism.

For an F-Logic program, which is a conjunction of F-facts and F-rules, we try to build a suitable *Herbrand model* by forward propagation. We expand the set of F-facts by repeatedly applying F-rules and inheritance and hence inferring new F-facts, until a stable state is reached. This set of F-facts is the model for this F-Logic program, i.e., the minimal set of F-facts that satisfies the program. Finally an F-query can be answered using this model.

While the current forward chaining approach facilitates interaction of deduction and inheritance, we might need a more sophisticated goal-driven mechanism for efficiency reasons in the future.

In Section 3.1 we show our representation for F-atoms and F-molecules and their decomposition. We illustrate deduction by F-rules (Section 3.2) and the similar mechanism for answering F-queries. We describe inference by structural and behavioural inheritance (Section 3.3). Optional check for type correctness w.r.t. signature expressions is explained (Section 3.4). Finally the effects of interaction between F-rules and inheritance are discussed with several examples (Section 3.5).

Complete sources and examples for our implementation (for SICStus Prolog [8]) are available [1].

3.1 F-Logic Facts in CHR

For F-facts we allow only ground terms. For convenience, F-molecules are supported and decomposed to F-atoms, so we describe F-atoms at first.

Our F-Logic syntax for CHR is intended to be visually as close as possible to the F-Logic notation proposed by Kifer [6]. The three categories of F-atoms are represented as CHR constraints: `a..b` for *class membership*, `a::b` for a *subclass relation*, and `a~(b)` for an *object atom*; data expressions are encoded as `m->>o` and `m*->>o`, signature expressions as `m=>>o` and `m*=>>o` (cf. Table 1), this is a total of six F-atom types. We call the components of F-atoms *arguments*, e.g., `a`, `b`, `m` and `o` in the last F-atoms. We define Prolog infix operators with adapted precedences.

Table 1. F-atoms in CHR syntax

F-Logic Syntax	CHR-Syntax
$a : b$	`a..b`
$a :: b$	`a::b`
$a[m \twoheadrightarrow x]$	`a~(b->>c)`
$a[m \bullet\!\!\twoheadrightarrow x]$	`a~(b*->>c)`
$a[m \Rightarrow\!\!\!\Rightarrow x]$	`a~(b=>>c)`
$a[m \bullet\!\!\Rightarrow\!\!\!\Rightarrow x]$	`a~(b*=>>c)`

The handler removes duplicate F-facts to prevent infinite computations by cyclic rules, e.g., $A : B \longleftarrow A : B$.

F-molecules are decomposed into F-atoms (cf. Table 2), e.g., the molecule `a..c~[m*->>[[b1,b2]..c], m*=>>c]` is converted to the atoms `a..c`, `b1..c`, `b2..c`, `a~(m*->>b1)`, `a~(m*->>b2)`, and `a~(m*=>>c)`.

Table 2. Decomposition of F-molecules into F-atoms

F-molecule	F-atoms
`a~[b,c]`	`a~(b), a~(c)`
`[a,b]..c`	`a..c, b..c`
`[a,b]::c`	`a::c, b::c`
`a..b~(c)`	`a..b, a~(c)`
`a::b~(c)`	`a::b, a~(c)`
`a~(b->>[c1,c2])`	`a~(b->>c1), a~(b->>c2)`
`a~(b->>c~(d))`	`a~(b->>c), c~(d)`
`a~(b->>c..d)`	`a~(b->>c), c..d`
`a~(b->>c::d)`	`a~(b->>c), c::d`

These are some of the CHR rules dealing with decomposition:

```
O~[H|T]        <=> O~H, O~T.
```

```
O~(M->>[H|T]) <=> O~(M->>H), O~(M->>T).
O~(M->>V..C)  <=> O~(M->>V), V..C.
```

The first CHR rule handles objects with multiple expressions, the second methods with multiple values, and the last decomposes nested expressions.

3.2 F-Rule Application and F-Queries

F-Rule Application An F-Rule `R` is represented as `H <-- B` with *head* `H` being an F-atom and *body* `B` being an expression with conjunctions and disjunctions of F-atoms, the arguments of F-atoms may be variables or ground terms:

`R` := `H <-- B`
`H` := F-atom
`B` := F-atom | `B or B` | `B and B`

At first, the F-rules are transformed, s.t. each body is a *conjunction*. Then, for each F-rule, we search for all variable instantiations that satisfy its body, i.e., that the corresponding F-facts do exist. At last, we apply each satisfying instantiation on the corresponding head and propagate it. The deduced F-facts however must be ground terms.

First, the handler converts expressions in the body to disjunctive normal form (DNF), and splits the F-rule for every part of the disjunction, so that each body is a conjunction of F-atoms:

```
Head <-- B and (C or D) <=> Head <-- (B and C) or (B and D).
Head <-- (B or C) and D <=> Head <-- (B and D) or (C and D).
Head <-- B or C         <=> Head <-- B, Head <-- C.
```

For storing variable bindings, we introduce a *partially satisfied rule*: the CHR constraint `frule/3` for F-rules with equality restrictions.

Let H, B_1,...,B_n be F-atoms, X_1,...,X_m be ground terms or variables and Y_1,...,Y_m be ground terms, then

```
frule(H<--[B1,...,Bn], [X1,...,Xm],[Y1,...,Ym])
```

is defined by

$$H \longleftarrow B_1 \textit{ and ... and } B_n \textit{ and } X_1 = Y_1 \textit{ and ... and } X_m = Y_m$$

For every F-rule, we convert the conjunction in the body to a list and propagate an `frule` without equality restrictions.

For example, the F-rule `H <-- B and (C or D)` is pre-processed by the following steps (let `H`, `B`, `C` and `D` be F-atoms):

- `H <-- B and (C or D).`
- `H <-- (B and C) or (B and D).`
- `H <-- B and C, H <-- B and D.`
- `frule(H<--[B,C], [],[]), frule(H<--[B,D], [],[]).`

Second, we search matching F-facts for every F-atom in the `frule` body and store the corresponding variable bindings. One CHR rule deals with each of the six types of F-atoms, e.g., consider this code fragment:

```
frule(Head <-- [Q1..Q2       |BR], X,V), P1..P2
  ==> X2=[Q1,Q2|X],    V2=[P1,P2|V],    subsumes_chk(X2,V2) |
      frule(Head <-- BR, X2,V2).
frule(Head <-- [Q1~(Q2*->>Q3)|BR], X,V), P1~(P2*->>P3)
  ==> X2=[Q1,Q2,Q3|X], V2=[P1,P2,P3|V], subsumes_chk(X2,V2) |
      frule(Head <-- BR, X2,V2).
```

Matching of the first piece of the F-rule body and an F-fact is possible if the F-atoms are of the same type and its arguments can be unified taking into account the previous variable bindings. We test for unification with the Prolog predicate `subsumes_chk`[1]. In the following examples, `frule` R and F-fact F can not interact as `subsumes_chk` fails (with `H` replaced by a proper head):

- Ground term should equal different ground term:
  R=`frule(H<--[X..goodguy], [],[])`, F=`jayne..badguy`.
  Here `subsumes_chk([X,goodguy],[jayne,badguy])` fails.
- Same variable should equal different ground terms:
  R=`frule(H<--[X~(likes->>X)], [],[])`, F=`kaylee~(likes->>simon)`.
  Here `subsumes_chk([X,likes,X],[kaylee,likes,simon])` fails.
- Previously bound variable should equal different ground term:
  R=`frule(H<--[X..goodguy], [X],[mal])`, F=`kaylee..goodguy`.
  Here `subsumes_chk([X,goodguy,X],[kaylee,goodguy,mal])` fails.

If the test for matching succeeds, we propagate a new `frule`-constraint with the first part of the body removed, but equality restrictions[2] added.

We add the elements, i.e., object, method, and return value, of the F-rule's body piece to the list in the second argument of `frule`. In parallel, we add the elements of the F-atom, that matches the body, to the list in the third argument of `frule`. As the list in the third argument is ground, `subsumes_chk` can check for unification.

We use CHR *propagation* instead of CHR *simplification* rules to get *all* variable bindings from different matching F-facts.

Compared to its ancestor, the list of F-atoms in the body of an `frule` is decreased by one. For an empty body the `frule` is satisfied solely by the equality restrictions, so finally we apply them on the head and propagate it as a new F-fact:

[1] `subsumes_chk(A,B)` tests if A subsumes B, i.e., whether an instantiation of A exists, so that A=B, e.g., it succeeds for `A=[X,X,Y,c]`, `B=[a,a,c,c]`, but fails for `A=a`, `B=X` or `A=a`, `B=b` or `A=[X,X]`, `B=[a,b]`. If B is a ground term like in our case, a test for subsumption is also a test for unification.

[2] Equality restrictions are guaranteed contradiction-free and either of the useful form $X = a$ or the trivial form $a = a$.

```
frule(Head <-- [], [X1|XR],[V1|VR])  <=>
   instantiate(X1=V1,Head,Head2) | frule(Head2 <-- [], XR,VR).
frule(A~B  <-- [], [],[])            <=>     A~B.
```

The user-defined predicate `instantiate(X=V, Term, Term2)` replaces all occurrences of `X` in `Term` by `V` and copies the result to `Term2`. Note that the variables in the term are instantiated manually by digging into the term structure. We do not use build-in unification to leave the variables in the F-rules untouched.

F-Queries An F-Query `? Q` is a shortcut for the pseudo F-rule `query <-- Q`. The same CHR rules are used for F-queries, except for the third and last step, i.e., dealing with empty bodies:

```
frule(query <-- [], X,V) <=> frule(query([]) <-- [], X,V).
frule(query(L) <-- [], [X1|XR],[V1|VR]) <=> var(X1) |
   frule(query([X1=V1|L]) <-- [], XR,VR).
frule(query(L) <-- [], [X1|XR],[_|VR])  <=> ground(X1) |
   frule(query(L) <-- [], XR,VR).
frule(query(L) <-- [], [],[])           <=>
   remove_duplicates(L,L2), reverse(L2,L3) | result(L3).
```

Ignoring trivial equality constraints (third CHR rule), we collect the remaining equalities (second CHR rule), remove duplicates, and store the final list in the constraint `result/1` (fourth CHR rule).

A separate `result` is returned for each valid set of variable instantiations for each disjunctive part of the F-query. A single `result` contains one value for each variable. Hence the answer is in disjunctive normal form, e.g.: `result([X=a])`, `result([X=b,Y=c]` corresponds to $X = a$ *or* $(X = b$ *and* $Y = c)$.

Consequently for an F-query without solution no `result` is returned. For a valid F-query without variables we return the unrestricted `result([])`.

```
kaylee..goodguy, simon..goodguy,
X~(likes->>Y) <-- X..goodguy and Y..goodguy,
? Who~(Meth->>kaylee).
```

The following CHR constraints are derived from this F-Logic program:

```
kaylee~(likes->>simon), kaylee~(likes->>kaylee),
simon~(likes->>simon), simon~(likes->>kaylee),
result([Who=kaylee, Meth=likes]),
result([Who=simon, Meth=likes]).
```

Therefore the answer to the F-query is:

$$(Who = kaylee \; and \; Meth = likes) \; or \; (Who = simon \; and \; Meth = likes)$$

3.3 Inheritance

While *structural inheritance*, responsible for is-a atoms and inheritable signature expressions, takes place in any case, *behavioural inheritance* for inheritable data expressions may be blocked by overriding.

On the one hand, structural inheritance is implemented by four simple CHR rules:

```
O..C1,  C1::C2   ==> O..C2.
C1::C2, C2::C3   ==> C1::C3.
O..C, C~(M*=>>V) ==> O~(M=>>V).
S::C, C~(M*=>>V) ==> S~(M*=>>V).
```

On the other hand, behavioural inheritance is more complicated: A data expression must not be inherited immediately, but must allow F-facts, that are defined afterwards or are derived by F-rules, to override the method. So the user must explicitly trigger inheritance by completing an F-Logic program with the constraint `inheritance`.

We create a `candidate` constraint for every potentially inheriting method. Afterwards we remove candidates that originate from inherited methods or are overridden by a more specific candidate or a method definition. The trigger then selects one candidate for actual inheritance.

For inheritance to both subclasses and members, we create a `candidate` with type, destination and source object, and method identifier:

```
O..C, C~(M*->>_) ==> candidate(member,O,C,M).
S::C, C~(M*->>_) ==> candidate(class, S,C,M).
```

Note that several identical `candidate`s are created for methods with multiple values, however these duplicates are removed by a CHR rule.

We do not allow inherited methods to be inherited further. This important point is explained at the end of the section. We identify these troublesome methods `c~(m*->>_)` by an existing `inherit(class,c,_,m)` (`inherit` has the same structure as `candidate`, but indicates *certain* inheritance; see below). We remove the corresponding `candidate`:

```
inherit(class,C,_,M) \ candidate(_,_,C,M) <=> true.
```

Two `candidate`s are called *competing*, if they are identical except for the source object. For two competing `candidates` the less specific, i.e., whose source object is a superclass of the other, is removed:

```
C1::C2, candidate(T,O,C1,M) \ candidate(T,O,C2,M) <=> true.
```

If the source objects of two competing candidates are incomparable, we might have multiple inheritance, which is indicated by a warning. Candidates overridden by an appropriate method definition of the destination object are removed:

```
O~(M->>_)  \ candidate(member,O,_,M) <=> true.
O~(M*->>_) \ candidate(class, O,_,M) <=> true.
```

The trigger `inheritance` selects a remaining `candidate` for actual inheritance and marks it with `inherit`:

```
inheritance, candidate(T,O,C,M) ==> inherit(T,O,C,M).
```

The constraint `inherit` has the effect, that the complete method is inherited in any case, probably consisting of multiple expressions for multiple values. Further competing candidates are overridden by newly inherited method expressions. Therefore inheritance from competing `candidates` selects one "winner" non-deterministically. To inhibit the selection of multiple competing `candidate`s we use CHR refined semantics and place the previous CHR rule at the end of the inheritance block. The CHR rules for actually doing the inheritance are placed before:

```
inherit(member,O,C,M), C~(M*->>V) ==> O~(M->>V).
inherit(class, S,C,M), C~(M*->>V) ==> S~(M*->>V).
```

We complete the section with an example:

```
mal..goodguy, mal..badguy, [goodguy,badguy]::person,
badguy~(attitude*->>aggressive),person~(attitude*->>friendly).
```

We clearly expect `attitude` to be inherited from `person` to `goodguy`, and from `badguy` to `mal`. What will happen in our handler?

`mal..person` is inherited structurally, and the following `candidate`s are created:

```
candidate(member,mal,badguy,attitude),
candidate(class,goodguy,person,attitude),
candidate(member,mal,person,attitude).
```

The `attitude` for `mal` could potentially be inherited from `badguy` or `person`, but because of being not so specific, the `candidate` for `person` is removed. Depending on the sequencing of the F-facts, inheritance from `person` to `goodguy` may occur first, then we get these constraints:

```
candidate(member,mal,badguy,attitude),
inherit(class,goodguy,person,attitude),
goodguy~(attitude*->>friendly),
candidate(member,mal,goodguy,attitude).
```

Now `mal` could potentially inherit from `goodguy` or `badguy`. But inheriting from `goodguy` is in fact inheriting indirectly from `person`, which was rejected in favour of `badguy`. Because of inheritance for is-a atoms, inheriting previously inherited methods is never necessary. Therefore we do not permit it, we detect this by means of `inherit` and remove `candidate(member,mal,goodguy,attitude)`. Finally we have:

```
goodguy~(attitude*->>friendly), mal~(attitude->>aggressive).
```

If we explicitly define `goodguy~(attitude*->>friendly)`, however, we get a multiple inheritance conflict and either `friendly` or `aggressive` is inherited for `mal`'s `attitude`.

3.4 Type Checking

For every pair of corresponding data and signature expression (`O~(M->>V)`, `O~(M=>>Type)` and `O~(M*->>V)`, `O~(M*=>>Type)`, respectively), we expect the type restriction `V..Type` to be fulfilled for type correctness. If this is not the case, the program aborts with an error message.

To allow the membership relation *after* data and signature expression, type checking should not occur immediately, so the user has to trigger type checking with `typecheck` if desired. An arbitrary number (including zero) of signature expressions is allowed per data expression, for type correctness every type restriction must be fulfilled.

Type checking is implemented by four CHR rules:

```
O~(M->>V),  O~(M=>>Type)  ==> expect(V..Type).
O~(M*->>V), O~(M*=>>Type) ==> expect(V..Type).
V..Type \ expect(V..Type) <=> true.
typecheck,expect(_) <=> write('error: typecheck failed'),fail.
```

The following example is type correct:

```
friendly..normal, aggressive..bad,
jayne~[attitude->>friendly, attitude=>>normal],
badguy~[attitude*->>aggressive, attitude*=>>bad].
```

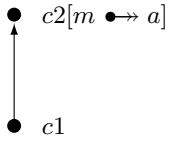
But if we add `jayne..badguy`, the signature `jayne~(attitude=>>bad)` is inherited structurally, therefore *additionally* `friendly..bad` is expected and type checking fails.

3.5 Interaction between F-Rule Application and Inheritance

The interaction of inheritance and deduction leads to several anomalies. We implement Kifer's original fixpoint computation scheme for inheritance [6] and discuss the behaviour of our handler w.r.t. three typical problematic examples from Yang [11]. On the left of the figures we illustrate the object hierarchy, on the right we show the F-Logic program. An arrow from node c1 to another node c2 indicates that c1 is a subclass of c2.

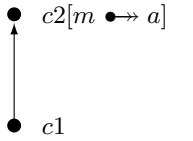
In Figure 1 our handler inherits `(m*->>a)` from `c2` to `c1`. Therefore through F-rule application `c2~(m*->>b)` is inferred, which leads to further inheritance. Altogether the following F-facts are inferred:

```
c1~(m*->>a), c2~(m*->>b), c1~(m*->>b).
```




```
c1::c2,
c2~(m*->>a),
c2~(m*->>b) <-- c1~(m*->>a).
```

Fig. 1. Superclass changed by rule




```
c1::c2,
c2~(m*->>a),
c1~(m*->>b) <-- c1~(m*->>a).
```

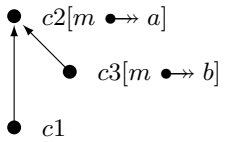
Fig. 2. Subclass changed by rule

Our implementation is correct for Yang's first problem.

In Figure 2 only the head of the F-rule is different. Our handler again inherits the method expression from `c2` to `c1`, so the F-rule is applied and `c1~(m*->>b)` is propagated. Therefore `c1` has an inherited and a deduced method expression:

```
c1~(m*->>a), c1~(m*->>b).
```

The answer of our implementation differs from Yang's, that interprets a deduced expression as equivalent to a local definition, so it should override the method and inhibit the previous inheritance, and therefore undermine the reason for the preceding F-rule application. The only reasonable solution would be leaving both F-facts undefined.




```
c1::c2, c3::c2,
c2~(m*->>a),
c3~(m*->>b),
c1::c3 <-- c1~(m*->>a).
```

Fig. 3. Hierarchy changed by rule

In Figure 3 our handler inherits the method expression from `c2` to `c1`, so by F-rule application, `c1::c3` is added and `c3` becomes more specific than `c2`. Nevertheless inheritance from `c3` to `c1` is not possible, because it is blocked by `c1~(m*->>a)`. Therefore we infer:

```
c1~(m*->>a), c1::c3.
```

Following Yang's interpretation, however, the method expression for `c1` should be inherited from the more specific `c3` *instead* of `c2` and therefore undermine the reason for the preceding F-rule application. Consequently all three F-facts should be left undefined:

```
c1~(m*->>a), c1::c3, c1~(m*->>b).
```

As mentioned in the last two examples, our current handler's fixpoint computation does not deal with all interactions between inheritance and F-rules in a reasonable way. As Yang's suggested implementation of inheritance [11] relies on general logic programming, a CHR implementation to handle it, will probably be more difficult and left to future work.

4 Conclusion

We follow the trend to use CHR as a general purpose programming language. We implemented the main features of F-Logic in CHR for refined semantics, including object-oriented syntax, F-Logic deduction, inheritance, and type-checking. Compared to the available FLORA-2 system, which uses general logic programming and well-founded semantics, our CHR implementation of (parts of) F-Logic is concise. Complete sources and examples for our implementation (for SICStus Prolog [8]) are available [1].

Implementing F-Logic in CHR is a first step towards a hybrid object-oriented rule-based constraint language. Currently, we investigate how the intricate interaction of F-rule deduction and (non-monotonic) inheritance can be implemented in CHR. A full F-Logic implementation in CHR should be simpler and thus easier to use than the currently available FLORA-2 engine. Considering all (F-Logic related) aspects of FLORA-2, including its performance scalability, our concise and purely data driven forward chaining approach might need to be revised.

References

1. CHR program sources for this paper (SICStus Prolog 3.11), May 2006. http://www.informatik.uni-ulm.de/pm/index.php?id=134.
2. Anthony J. Bonner and Michael Kifer. Transaction logic programming (or, a logic of procedural and declarative knowledge). Technical Report CSRI-323, Computer Systems Research Institute, University of Toronto, 1995.
3. Weidong Chen, Michael Kifer, and David S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.
4. Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
5. Michael Kifer. Nonmonotonic reasoning in FLORA-2. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *LPNMR*, volume 3662 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005.
6. Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *J. ACM*, 42(4):741–843, 1995.

7. ROARS Reused Oriented Automated Reasoning Software (project web page), 2006. http://www.cin.ufpe.br/ jr/mysite/RoarsProject.html.
8. SICStus Prolog web page, 2006. SICStus 3.11.2, http://www.sics.se/sicstus/.
9. XSB Prolog web page, 2006. XSB 2.7.1, http://xsb.sourceforge.net/.
10. Guizhen Yang. *A Model Theory for Nonmonotonic Multiple Value and Code Inheritance in Object-Oriented Knowledge Bases.* PhD thesis, State University of New York at Stony Brook, USA, 2002.
11. Guizhen Yang and Michael Kifer. Well-founded optimism: Inheritance in frame-based knowledge bases. In Robert Meersman and Zahir Tari, editors, *CoopIS/DOA/ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 1013–1032. Springer, 2002.
12. Guizhen Yang, Michael Kifer, Chang Zhao, and Vishal Chowdhary. *FLORA-2: User's Manual, Version 0.94 (Narumigata)*, 2005. http://flora.sourceforge.net/docs/floraManual.pdf.

Deriving Quasi-Linear-Time Algorithms from Union-Find in CHR
Extended Abstract

Thom Frühwirth

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

Abstract. The union-find algorithm can be seen as solving simple equations between variables or constants. With a few lines of code change, we generalise its implementation in CHR from equality to arbitrary binary relations. By choosing the appropriate relations, we can derive fast algorithms for solving certain propositional logic (SAT) problems as well as certain polynomial equations in two variables. While linear-time algorithms are known to check satisfiability and to exhibit certain solutions of these problems, our algorithms are simple instances of the generic algorithm and have additional properties that make them suitable for incorporation into constraint solvers: From classical union-find, they inherit simplicity and quasi-linear time and space. By nature of CHR, they are anytime and online algorithms. They can be parallelised. They solve and simplify the constraints in the problem, and can test them for entailment, even when the constraints arrive incrementally, one after the other. We show that instances where relations are bijective functions yield precise and correct algorithm instances of our generalised union-find.

1 Introduction

Constraint Handling Rules (CHR) [6,8] is a logical constraint-based concurrent committed-choice programming language consisting of guarded rules that rewrite multisets of atomic formulas.

It was shown recently that the classical optimal *union-find* algorithm [15] is implementable in CHR with best-known *quasi-linear time* complexity [12,13]. This result is not accidental, since the paper [14] shows that a subset of the CHR language can simulate Turing and RAM machines in polynomial time, thus establishing that CHR is Turing-complete and, more importantly, that every algorithm can be implemented in CHR with best known time and space complexity, something that is not known to be possible in other pure declarative programming languages.

The union-find algorithm maintains disjoint sets under the operation of union. By definition of set operations, a union operator working on representatives of sets is an equivalence relation, i.e. we can view sets as equivalence

classes. Especially iff the elements of the set are variables or constants, union can be seen as equating those elements and giving an efficient way of finding out if two elements are equivalent (i.e., in the same set).

This extended abstract investigates the question if the union-find algorithm written in CHR can be generalised so that other relations than simple equations between two variables are possible without compromising efficiency.

Overview of the Paper. This extended abstract discusses the following topics in the next sections.

- Quasi-Linear Time Union-Find Algorithm
- Generalised Union-Find in CHR
- Example Instances: Boolean and Polynomial Equations in Two Variables
- Time and Space Complexity and Correctness of the Generalisation
- Conclusions and Future Work

We assume some familiarity with CHR [6,8] in this extended abstract.

2 The Union-Find Algorithm

In this section we follow the exposition of [12]. The classical union-find (also referred to as disjoint-set-union) algorithm was introduced by Tarjan in the seventies [15]. A classic survey on the topic is [9]. The algorithm solves the problem of maintaining a collection of disjoint sets under the operation of union. Each set is represented by a rooted tree, whose nodes are the elements of the set. The root is called the *representative* of the set. The representative may change when the set is updated by a union operation. With the algorithm come three operations on the sets:

- `make(X)`: create a new set with the single element `X`.
- `find(X)`: return the representative of the set in which `X` is contained.
- `union(X,Y)`: join the two sets that contain `X` and `Y`, respectively (possibly changing the representative).

A new element must be introduced exactly once with `make` before being subject to `union` and `find` operations. To find out if two elements are in the same set already, i.e. to check entailment, one finds their representatives and checks them for equality, i.e. checks `find(X)=find(Y)`.

2.1 Implementing Union-Find in CHR

The following CHR program in concrete ASCII syntax implements the operations and data structures of the naive union-find algorithm without optimisations. In the naive algorithm, these three operations are implemented as follows.

- `make(X)`: generate a new tree with the only node `X`, i.e. `X` is the root.
- `find(X)`: follow the path from the node `X` to the root of the tree. Return the root as representative.

– `union(X,Y)`: find the representatives of `X` and `Y`, respectively. To join the two trees, it suffices to `link` them by making one root point to the other root.

In the CHR implementation, the constraints `make/1`, `union/2`, `find/2` and `link/2` define the operations (functions are written in relational form), so we call them *operation constraints*. The constraints `root/1` and `->/2` (using infix notation) represent the tree data structure and we call them *data constraints*. We use the infix notation `->/2` to evoke the image of a directed arc, since it is often helpful for the understanding of the algorithm to imagine the tree as directed graph.

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

findNode @ X -> Y, find(X,R) <=> X -> Y, find(Y,R).
findRoot @ root(X), find(X,R) <=> root(X), R=X.

linkEq   @ link(X,X) <=> true.
link     @ link(X,Y), root(X), root(Y) <=> X -> Y, root(Y).
```

2.2 Optimised Union-Find

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

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

The second optimisation is *union-by-rank*. It keeps the tree shallow by pointing the root of the smaller tree to the root of the larger tree. *Rank* refers to an upper bound of the tree depth (tree height). If the two trees have the same rank, either direction of pointing is chosen but the rank is increased by one. With this optimisation, the height of the tree can be logarithmically bound.

The following CHR program implements the optimised classical union-find algorithm with path compression for find and union-by-rank [15].

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

findNode @ X -> Y, find(X,R) <=> find(Y,R), X -> R.
findRoot @ root(X,N), find(X,R) <=> root(X,N), R=X.

linkEq   @ link(X,X) <=> true.
linkLeft @ link(X,Y), root(X,RX), root(Y,RY) <=> RX>=RY |
```

```
              Y -> X, root(X,max(RX,RY+1)).
linkRight@ link(X,Y), root(Y,RY), root(X,RX) <=> RY>=RX |
              X -> Y, root(Y,max(RY,RX+1)).
```

When compared to the naive version `ufd_basic`, we see that `root` has been extended with a second argument that holds the rank of the root node. The `union/2` operation constraint is implemented exactly as for the naive algorithm. The rule `findNode` has been extended for path compression. By the help of the variable `R` that serves as a place holder for the result of the find operation, path compression is already achieved during the first pass, i.e. during the find operation. The `link` rule has been split into two rules, `linkLeft` and `linkRight`, to reflect the optimisation of union-by-rank: The smaller ranked tree is added to the larger ranked tree without changing its rank. When the ranks are the same, either tree is updated (both rules are applicable) and the rank is incremented by one.

3 Generalised Union-Find

The idea of generalising union find is to replace equations between variables by arbitrary binary relations. The operation `union` now asserts a given relation between its two variables, `find` finds the relation between a given variable and the root of the tree in which it occurs. The operation `link` includes the relation as well so that it is stored in the tree data constraint, i.e. the arcs in the tree are labelled by relations now. In the CHR implementation, the operation constraints `union`, `find`, `link` and the data constraint `->` get an additional argument to hold the relation.

We also need some standard *operations on relations* from relational algebra that are implemented by constraints as follows (where we use relational notation and id is the identity function, i.e. equality):

- `compose`(r_1, r_2, r_3) iff $r_1 \circ r_2 = r_3$
- `invert`(r_1, r_2) iff $r_1 = r_2^{-1}$
- `equal`(r_1) iff $r_1 = id$

The following code extends the CHR implementation of optimal union-find by additional arguments and by additional constraints on them. These additions are emphasised for clarity. Our implementation in Sicstus Prolog CHR is available at `www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/more/ufe.pl`.

```
make     @ make(X) <=> root(X,0).
union    @ union(X,XY,Y) <=> find(X,XA,A), find(Y,YB,B),
                             combine(XA,YB,XY,AB), link(A,AB,B).

findNode @ X-XY->Y, find(X,XR,R) <=> find(Y,YR,R),
                             compose(XY,YR,XR), X-XR->R.
findRoot @ root(X,N), find(X,XR,R) <=> root(X,N), equal(XR), X=R.
```

```
linkEq   @ link(X,XX,X) <=> equal(XX).
linkLeft @ link(X,XY,Y), root(X,RX), root(Y,RY) <=> RX>=RY |
           invert(XY,YX), Y-YX->X, root(X,max(RX,RY+1)).
linkRight@ link(X,XY,Y), root(Y,RY), root(X,RX) <=> RY>=RX |
                         X-XY->Y, root(Y,max(RY,RX+1)).
```

The operation constraint `union(X,XY,Y)` now means that we enforce relation `XY` between `X` and `Y`. The operation `find` still returns the root for a given node, but also the relation that holds between the node and the root. From these three relations, `combine` derives the relation `AB` that must hold between the roots that are to be linked.

The `combine` operation can be defined in CHR as follows:

```
combine(XA,YB,XY,AB) <=>
    compose(XY,YB,XB),
    invert(XA,AX),
    compose(AX,XB,AB).
```

```
X -- XA* -- A
|           |
XY         AB?
|           |
Y -- YB* -- B
```

The crude graphics on the right of the CHR code shows the relations between the four relations that are arguments of `combine`. It is a so-called *commutative diagram*. The question mark after `AB` reminds us that this relation is the one that `combine` computes from the other three. The starred relations `*` remind us that `find` computes these relations by repeated composition of the relations on the path of the tree. Overall, given the relations between `X` and `Y`, `X` and `A`, `Y` and `B`, `combine` computes the relation between `A` and `B` that will replace the relation `XY` in the tree representation.

4 Instance of Boolean Equations

With our generalised union-find algorithm, we can solve inequations between Boolean variables (propositions), i.e. certain 2-SAT problems. This instance features a finite domain and a finite number of relations. In the CHR implementation, the relations are `eq` for $=$ and `ne` for $\neq$, and the truth values are `0` for false and `1` for true. The operations on relations can be defined by the following facts:

```
compose(eq,R,R).
compose(R,eq,R).
compose(ne,ne,eq).
```

```
invert(X,X).

equal(eq).
```

Here is a very simple example of a query for Booleans. Note that we introduce `0` and `1` by `make` and add `union(0,ne,1)` to enforce that they are distinct. This suffices to solve this type of Boolean inequations and to simplify them thanks to union-find such that the relation (`eq` or `eq` or none) between variables can be found in quasi-constant time.

```
?- make(0),make(1),union(0,ne,1),
   make(A),make(B),union(A,eq,B),union(A,ne,0),union(B,eq,1).
root(A,2), B-eq->A, 0-ne->A, 1-eq->A.
```

The result of the query shows that `A` is also equal to `1`.

Related Work. It is well known that 2-SAT (conjunctions of disjunctions of at most two literals) [1] and Horn-SAT (conjunctions of disjunctions with at most one positive literal, i.e. propositional Horn clauses) [3,5,11] can be checked for satisfiability in linear time.

The class of Boolean equations and inequations we can deal with is a proper subset of 2-SAT, but not of Horn-SAT, since `A ne B` $\Leftrightarrow$ (`A` $\vee$ `B`) $\wedge$ ($\neg$`A` $\vee$ $\neg$`B`).

These two classical linear-time SAT algorithms assume that the graph is initially known, because it has to be traversed along its edges. The algorithms only check for satisfiability and can report one possible solution, but they do not simplify or solve the given problem in a general way.

The 2-SAT algorithm translates a given problem into a directed graph where arcs are the implications that are logically equivalent to the individual clauses in the problem. It then relies on a linear-time preprocessing of the given graph to find is maximal strongly connected components in reverse topological order. Then truth values are assigned to components by assigning them to all nodes in that component. Respecting the topological order, truth values are propagated through the components.

In contrast, our generalised union-find algorithm is an online algorithm and incremental in the sense of constraint logic programming. It can find the relation between two given variables in amortised quasi-constant time and space. It produces a simple normal form that has the same size of the original problem. By using find, the results can be normalised in quasi-linear time. It can be used for ask and tell, for assertion and entailment testing of constraints and is thus well-suited to be used in constraint solvers. It can even be run in parallel on variable-disjoint parts of the problem (that is, unions that share variables must be processed in sequential order). A more parallel version of union-find in CHR is discussed in [7].

Our Boolean algorithm instance can be integrated into a Boolean constraint solver. For example, the classical Boolean solver in CHR is based on value (unit) propagation, e.g. `and(X,Y,Z) <=> X=0 | Z=0`, and propagation of equalities, e.g. `and(X,Y,Z) <=> X=Y | Y=Z`. It can be now extended by propagation of inequalities, e.g. `and(X,Y,Z) <=>` $X\bar{Y}$ `| Z=0` and `and(X,Y,Z) <=>` $X\bar{Z}$ `| X=1,Y=0,Z=0`.

Can we extend our algorithm instance of generalised union-find to deal with 2-SAT? As put to use in the classical algorithm, any disjunction in two variables, `A` $\vee$ `B` can be written as implication $\neg$`A` $\rightarrow$ `B`. Since we can get negation using an auxiliary variable `A ne negA`, we just would have to introduce the relation $\rightarrow$ (that corresponds to a total non-strict order $\leq$ on the truth values). But the implication relation looses too much information when composed. For example, given a tree `B-`$\leq$`->A, C-`$\leq$`->A`, `B` and `C` can be arbitrarily related. So if one adds

`union(B,eq,C)` it has no effect on the tree, and thus the information that `B eq C` is lost.

5 Instance of Linear Polynomials

Another instance of our generalised union-find algorithm deals with linear polynomial equations in two variables. It features an infinite domain and an infinite number of relations. The CHR data constraint `X-A|B->Y` (with `A`$\neq$`0`) now means `X=A*Y+B`. Note that these type of equations can be interpreted as functions. The operations on relations are defined as follows:

```
compose(A|B,C|D,A*C|A*D+B).
invert(A|B,1/A|-B/A).                    equal(1|0).
```

Again, a small example illustrates the behaviour of this instance.

```
?- make(X),make(Y),make(Z),make(W),
   union(X,2|3,Y),union(Y,0.5|2,Z),union(X,1|6,W).
root(X,1), Y-0.5|-1.5->X, Z-1.0|-7.0->X, W-1.0|-6.0->X.
```

Note that by the generic `linkEq` rule, `link(X,1|0,X)` will succeed but all other equations involving only one variable will fail. While this is as expected for e.g. `link(X,1|1,X)`, the equation `link(X,2|1,X)` has a solution `X=-1`. Indeed, in our program, failure will occur whenever a variable is fixed, i.e. determined to take a unique value. Our algorithm succeeds exactly when the set of equations has infinitely many solutions.

We now slightly modify the program code of the instance of our generalised union-find algorithm in order to introduce concrete numeric values and to solve for determined variables. Since there infinitely many numbers, we express them in terms of a single number, `1`. To make sure that the number `1` always stays the root, so that it can be always found by the find operation, we add `root(1,`∞`)` instead of `make(1)` to the beginning of a query. We replace the `linkEq` rule by the two rules. The first restricts applicability of the generic `linkEq` rule to the case where `A=1`, the second rule applies to equations that determine their variable and normalises the equation such that the coefficient is `1` and the second occurrence of the variable is replaced by the value `1`.

```
linkEq1 @ link(X,A|B,X) <=> A=:=1 | B=:=0.
linkEq2 @ link(X,A|B,X) <=> A=\=1 | link(X,1|B/(1-A)-1,1).
```

Note that there is a subtle point about these two rules: `X` may be the value `1`, and in that case the execution of `link(X,1|B/(1-A)-1,1)` in the right hand side of rule `linkEq2` will use rule `linkEq1` to check if `B/(1-A)-1` is zero (which holds if `B=1-A`).

The following tiny examples illustrate the behaviour of these two rules (∞ is chosen to be `9`):

```
?- root(1,9), make(X),make(Y),
   union(X,2|3,Y),union(X,4|1,1).
root(1,9), X-4|1->1, Y-0.5|-1.5->X.

?- root(1,9), make(X),make(Y),union(X,4|1,1),union(X,2|3,Y).
root(1,9), X-4|1->1, Y-2|-1->1.
```

We may add another rule that propagates values for determined variables down the tree data structure and so binds determined variables:

```
X-A|B->N <=> number(N) | X=A*N+B.

?- root(1,9), make(X),mak e(Y),
   union(X,2/3,Y),union(X,4/1,1).
root(1,9), X=5, Y=1.
```

Related Work. [2] gives a linear time algorithm that shares many principles with ours, but is more complicated. Equations correspond to directed arcs in a graph. Like the 2-SAT algorithm [1], it computes maximal strongly connected components. Inside each component, a modification of any linear-time spanning tree algorithm can be used to simplify the equations. The overall effect is the same as with our algorithm, and the algorithm is similar on the components, especially if Kruskal's algorithm [10] for spanning trees is used which relies on union-find. However, our algorithm is simpler and more general in its applicability. It does not need to compute strongly connected components or spanning tress, it directly uses union-find and moreover is incremental and parallelisable.

6 Complexity and Correctness

We want to show that our algorithm is a canonical extension of the optimised union-find algorithm in CHR. In other words, if we instantiate our algorithm to the case where the only relation is `=`, we get back the original program.

We first discuss *complexity* of our algorithm.

Theorem 1 (Complexity). Our algorithm has the same time and space complexity as the original algorithm if the operations on relations take constant time and space.

Proof Sketch. Any computation in our generalised algorithm can be mapped into a computation of the original union-find algorithm or it fails.

In the generalisation, we added arguments for the relations to existing CHR constraints that are variables and constraints on these variables only. The additional variables on the left hand side of each rule are all distinct and the guards have not been changed. The additional constraints on the right only constrain the new variables. In CHR, additional constraints can cause failure (inconsistency) or make more rules applicable, but never less. Since the new constraints are on new variables only, and these are not checked by the left hand sides and guards of rules, the applicability of the rules remains unchanged. So the only change

is that a rule application could cause failure where the computation proceeded before (this applies to the `linkEq` rule).

More formally, we construct a mapping function that removes the additional arguments and additional built-in constraints. The claim is then shown by induction on length of the derivation and case analysis of the rules applicable in a derivation step. □

Next we show that our implementation is *correct*, if the involved relations are *bijective functions*. In that case, the composition operation is precise enough in that it allows to derive any of the three involved relations from the other two.

Intuitively the reason why we cannot work with arbitrary relations is that given n variables, there are up to $n(n-1)$ binary relations between different variables, but we can only represent $n-1$ of them in the tree data structure of the union-find algorithm. That means that all the other possibly existing relations must be computable from these $n-1$ relations.

In the following we assume w.l.o.g. that the domains of a variable include only those values for which the relations in which it occurs are defined. If the relation is a function this means that the function is total.

Definition 1. A function is *bijective* if for every y there is exactly one x such that $f(x) = y$, that is $f(x) = y \wedge f(u) = v \wedge (x = u \vee y = v) \rightarrow x = u \wedge y = v$.

Bijective functions are closed under inverse and composition.

While bijective functions may seem quite a strong restriction we remind the reader that permutations, isomorphisms and many other mappings (such as encodings in cryptography) are bijective functions and that most arithmetic functions are at least piecewise bijective, since they are piecewise monotone. Indeed, for a domain of size n, there exist $n!$ different bijective functions, i.e. more than exponentially many.

The identity function *id* is bijective. The relations `eq` and `ne` are bijective functions for domains with at most two values, and thus for Booleans. Linear polynomial equations in two variables can also be interpreted also bijective functions.

Theorem 2 (Correctness). The logical reading of the rules of our generalised union-find algorithm (for classical union-find, see [13]) is a consequence of a theory for the relations if these relations are bijective functions.

Proof Sketch. We prove by analysing the logical reading of the rules, where we replace `union`, `find`, `link` and `->` as intended by the binary relations between their variables (using infix notation), and the constraints for operations on relations by their definitions using functional notation. As usual, formulas are assumed to be universally closed.

```
(make)      make(X) ⇔ root(X,0).
(union)     (X XY Y) ⇔ ∃XA,A,YB,B,AB ((X XA A) ∧ (Y YB B) ∧
                        XA^-1∘XY∘YB=AB ∧ (A AB B))
```

```
(findNode)  (X XY Y) ∧ (X XR R) ⇔ ∃YR ((Y YR R) ∧
                          XY∘YR=XR ∧ (X XR R))
(findRoot)  root(X,N) ∧ (X XR R) ⇔ root(X,N) ∧ XR=id ∧ X=R

(linkEq)    (X XX X) ⇔ XX=id
(linkLeft)  RX>=RY ⇒ ((X XY Y) ∧ root(X,RX) ∧ root(Y,RY) ⇔
               ∃YX (XY^-1=YX ∧ (Y YX X) ∧ root(X,max(RX,RY+1))))
(linkRight) RY>=RX ⇒ ((X XY Y) ∧ root(Y,RY) ∧ root(X,RX) ⇔
               (X XY Y) ∧ root(Y,max(RY,RX+1)))
```

Even though the logical reading in first order logical does not reflect the intended meaning of the `root` data constraint [13] (and a linear logic semantics is more faithful [4]), the logical reading suffices for our purposes.

Most rules lead to formulas that do not impose any restriction on the binary relations involved. The logical readings of `linkEq` and `findRoot` imply that the only relation that is allowed to hold between identical variables is the identity function *id*. The `findNode` rule, however, tells us a logical equivalence,

`(X XR R) ∧ (X XY Y) ⇔ (X XR R) ∧ (Y YR R)` where `XY∘YR=XR`,

that is not a tautology and restricts the involved relations. (For example, it does not hold for `≤=XR=YR=XY` even though $\leq \circ \leq = \leq$.)

This conditions is obviously satisfied if the involved relations are bijective functions, because then, for any value given to one of the variables, the values for the other two variables are uniquely determined on both sides of the logical equivalence and there cannot be another triple of values that has any of the values in the same component. □

To further illustrate the Correctness Theorem, let `XR = id`. Then the above condition simplifies to: `(X XY Y) ⇔ (Y YR X)` where `XY∘YR=id`. This means that for any relation g that takes the place of `XY` or `YR`, there must be a right and a left inverse, i.e. $g \circ g^{-1} = g^{-1} \circ g = id$. This is the case if the relations are bijective functions. As a counter-example, take the function g that is defined by $g(a)=c, g(b)=c$ and its inverse g^{-1}. Now g is not bijective since for c there exists more than one value that yields it when g is applied to it. Also, g^{-1} is not a function. The composition $g^{-1} \circ g$ yields the universal relation that includes any pair taken from $\{a, b\}$. The composition $g \circ g^{-1}$ yields the pair $< c, c >$. But both should yield the identity function *id*.

Intuitively, we can replace any operation in our generalised union-find algorithm on a given bijective function f by a conjunction of operations on its individual values using the classical union-find algorithm. That is, we claim $X \in D_X \wedge Y \in D_Y \wedge \texttt{union}(X, f, Y)$ is equivalent to $\bigwedge_{X \in D_X} \texttt{union}(X, f(Y))$, where D_V denotes the domain of values for variable V.

7 Conclusions

In this extended abstract we have presented work in progress about extending the applicability of union-find implemented in CHR. We saw that the general-

isation of the algorithm from maintaining equalities to certain binary relations (in particular bijective functions that admit precise composition) is straightforward in CHR and that the generalisation does not compromise quasi-linear time efficiency. We have implemented the generalisation and two instances, for equations and inequations over Booleans and for linear polynomial equations in two variables.

Our implementation in Sicstus Prolog CHR is available at
`www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/more/ufe.pl`.

While linear-time algorithms are known to check satisfiability and to exhibit certain solutions of these problems, our algorithms are simple instances of the generic algorithm and have additional properties that make them suitable for incorporation into constraint solvers: From classical union-find, they inherit simplicity, the possibility to both assert relations and test for entailed relations as well as quasi-linear time and space. By nature of CHR, they are anytime and online algorithms. They can be parallelised and solve and simplify the constraints in the problem, even when the constraints arrive incrementally, one after the other.

It remains to show that the proofs for complexity and correctness are as straightforward as claimed in this extended abstract. Future work will also try to extend the class of bijective functuions to other binary relations, and to investigate relationship with classes of tractable constraints. We also would like to investigate the potential tradeoff between efficiency and precision (i.e. by applying our generalised union-find to inequalities like $\leq$).

References

1. B. Aspvall, M.F. Plass, and R.E. Tarjan. A linear time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters*, 8:121–123, 1979.
2. Bengt Aspvall and Yossi Shiloach. A fast algorithm for solving systems of linear equations with two variables per equation. *Linear Algebra and its Applications*, 34:117–124, 1980.
3. Catriel Beeri and Philip A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Trans. Database Syst.*, 4(1):30–59, 1979.
4. Hariolf Betz and Thom Frühwirth. A linear-logic semantics for constraint handling rules. In P. van Beek, editor, *11th Conference on Principles and Practice of Constraint Programming CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 137–151. Springer, October 2005.
5. William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Log. Program.*, 1(3):267–284, 1984.
6. T. Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
7. T. Frühwirth. Parallelizing union-find in constraint handling rules using confluence. In M. Gabbrielli and G. Gupta, editors, *Logic Programming: 21st International Conference, ICLP 2005*, volume 3668 of *Lecture Notes in Computer Science*, pages 113–127. Springer, October 2005.

8. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.
9. Z. Galil and G. F. Italiano. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Comp. Surveys*, 23(3):319ff, 1991.
10. Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.
11. Michel Minoux. LTUR: a simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. *Information Processing Letters*, 29(1):1–12, September 1988.
12. T. Schrijvers and T. Frühwirth. Optimal union-find in constraint handling rules, programming pearl. *Theory and Practice of Logic Programming (TPLP)*, 6(1), 2006.
13. Tom Schrijvers and Thom Frühwirth. Analysing the CHR Implementation of Union-Find. In *19th Workshop on (Constraint) Logic Programming (W(C)LP 2005)*. Ulmer Informatik-Berichte 2005-01, University of Ulm, Germany, February 2005.
14. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The Computational Power and Complexity of Constraint Handling Rules. In *Second Workshop on Constraint Handling Rules, at ICLP05*, Sitges, Spain, October 2005.
15. Robert E. Tarjan and Jan van Leeuwen. Worst-case Analysis of Set Union Algorithms. *J. ACM*, 31(2):245–281, 1984.

Observable Confluence for Constraint Handling Rules

Gregory J. Duck[1], Peter J. Stuckey[1], and Martin Sulzmann[2]

[1] NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
University of Melbourne, 3010, AUSTRALIA
{gjd,pjs}@cs.mu.oz.au
[2] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg

Abstract. Constraint Handling Rules (CHRs) are a powerful rule based language for specifying constraint solvers. Critical for any rule based language is the notion of confluence, and for terminating CHRs there is a decidable test for confluence. But many CHR programs that in practice are confluent fail this confluence test. The problem is that the states that illustrate non-confluence are not reachable in practice. In this paper we introduce the notion of observable confluence, a weaker notion of confluence which takes into account whether states are observable. We show for an important class of non-confluent programs arising from Haskell type class programs with functional dependencies, that they are observable confluent.

1 Introduction

Constraint Handling Rules [3] (CHRs) are a powerful rule based language for specifying constraint solvers. Constraint handling rules operate on a global multiset (conjunction) of constraints. A constraint handling rule defines a rewriting from one multiset of constraints to another. There are two kinds of rewriting rules: simplification rules which replace one multiset of constraints by another, and propagation rules which when seeing a multiset of constraints matching some condition, add some more constraints to the global multiset.

Example 1. Consider the following CHR program:[3]

```
r1 @ f(int,bool,float)    <=> true.
r2 @ f(A,B1,C), f(A,B2,D) ==> B1 = B2.
r3 @ f(int,B,C)           ==> B = bool.
```

The first rule has name `r1`. It is a simplification rule. Whenever we see the constraint `f(int,bool,float)` in the global multiset we can replace it by `true` (representing the empty multiset). The second rule (named `r2`) is a propagation

[3] CHRs follow Prolog like notation, where identifiers starting with a lower case letter indicate predicates and function symbols, and identifiers starting with upper case letters indicate variables.

rule: whenever there exists two `f/3` constraints in the global multiset that share the same first argument, we can add the constraint that the second arguments must be identical. The third rule (`r3`) is another propagation rule, whenever we see a `f` constraint with first argument `int` we can enforce that the second argument is `bool`.

The program above results from the translation of type class constraints in Haskell [7] involving functional dependencies [5] to CHRs [4, 8]. Here is the original type class program.

```
class F x y z | x -> y
instance F Int Bool Float
```

Rule `r1` encodes the instance for `F`, that is we can prove their is an instance for `Int Bool Float`. Rule `r2` encodes the functional dependency, it requires that for any two `F` constraints if the first argument is the same, the second must also be the same (since it is functionally defined by the first). Rule `r3` encodes the combination of the instance rule and the functional dependency.

Unlike other rewriting systems, constraint handling rules allow propagation rules. Note that propagation rules do not delete anything from the global multiset that makes them eligible to be executed, they simply add new constraints. For this reason they would seem to trivially lead to non-termination. This is overcome in constraint handling rules by keeping a *history* of propagation rule firings, and preventing a second firing of a propagation rule on the same set of constraints.

A critical issue for any rule based language is the notion of confluence. Confluence enforces that each different possible rewriting sequence leads eventually to the same result. Confluent programs have a deterministic behaviour in terms of an input goal will always lead to the same answer. For terminating CHR programs there is a decidable test for confluence [1]. Unfortunately there are many (terminating) programs which are confluent in practice, but fail to pass the test.

The reason for this arises from the use of propagation rules and histories. The confluence test examines critical states, which are minimal states where two different rule firings are possible. In order to be minimal states, the propagation history is assumed to be as strong as possible, that is, it disallows any propagation rules that could possible fire except the two rules used to generate the critical state itself.

Example 2. Consider the CHR program of Example 1, and a critical state arising from rules `r1` and `r2`.

$$f(int, bool, float), f(int, B2, D)$$

Both rules `r1` and `r2` are applicable to the state and to ensure that it is minimal, we assume a propagation history which disallows the use of `r3` on either of the constraints appearing in the state.

The CHR program of Example 1 is not confluent, because this critical state can lead to 2 different results, depending on which of `r1` and `r2` is first used. This is illustrated by the derivations

$$\begin{array}{rl} & f(int, bool, float), f(int, B2, D) \\ \rightarrowtail_{r1} & f(int, B2, D) \end{array}$$

and

$$\begin{aligned} & f(int, bool, float), f(int, B2, D) \\ \rightarrowtail_{r2}\ & f(int, bool, float), f(int, bool, D), B2 = bool \\ \rightarrowtail_{r1}\ & f(int, bool, D), B2 = bool \end{aligned}$$

Two different states result. In one we know that $B2 = bool$ in the other we do not. Note that we cannot apply the rule `r3` to the state $f(int, B2, D)$ because the propagation history disallows this.

The above example illustrates that the program P of Example 1 is non-confluent. But in practice it is not possible to write a goal G which leads to two different answers using P. The reason is that goals always begin with an empty history. The critical state used above to illustrate non-confluence cannot actually occur in practice.

Example 3. Consider the state

$$f(int, bool, float), f(int, B2, D)$$

assuming that propagation history prevent `r3` firing on the second constraint. Thus `r3` must have fired already on the second constraint. This cannot have occurred since this would add the constraint $B2 = bool$ to the store, and it does not appear.

CHRs resulting from the translation of type class constraints in Haskell involving functional dependencies are an important class of programs since the soundness and completeness of type inference for Haskell with type classes and functional dependencies depends on their confluent behaviour [4, 8].

In this paper, we make the following contributions:

- We introduce the notion of *observable confluence* which captures the notion of programs where all the observable behaviour (derivations for goals with originally empty propagation histories) is confluent (Section 2.3).
- We show that for a class of CHRs arising from Haskell type class programs with functional dependencies the current notion of confluence is too limiting (Section 3).
- We provide for a general observable confluence result based on a operational correspondence condition between two sets of CHRs (Section 4).
- We verify that a certain class of non-confluent CHRs resulting from type class programs with functional dependencies are in fact observable confluent (Section 5).

In Section 2 we provide background material on CHRs and also introduce the notion of observable confluence. We conclude in Section 6.

2 Preliminaries

In this paper we study confluence under the *theoretical operational semantics* ω_t of CHRs [2]. The theoretical semantics is equivalent to the original semantics defined in [1] except that

– the original semantics treats simpagation rules as shorthand for a (logically) equivalent simplification rule; and
– the treatment of propagation histories is different.

First we define numbered constraints.

Definition 1 (Numbered Constraints). *A* numbered constraint *is a constraint c paired with an integer i. We write $c\#i$ to indicate a numbered constraint.* □

Sometimes we refer to i as the *identifier* (or simply ID) of the numbered constraint. This numbering serves to differentiate among copies of the same constraint.

Now we define an *execution state*, as follows.

Definition 2 (Execution State). *An* execution state *is a tuple of the form $\langle G, S, B, T\rangle_n^{\mathcal{V}}$ where G is a multiset of constraints, S is a set of numbered constraints, B is a conjunction of built-in constraints, T is a set of sequences of integers, $\mathcal{V}$ is the set of variables and n is an integer. Throughout this paper we use symbol 'σ' to represent an execution state.* □

We call G the *goal*, which contains all constraints to be executed. The CHR constraint *store* S is the set[4] of *numbered* CHR constraints that can be matched with rules in the program P. For convenience we introduce functions $chr(c\#i) = c$ and $id(c\#i) = i$, and extend them to sequences and sets of numbered CHR constraints in the obvious manner.

The *built-in constraint store* B contains any built-in constraint that has been passed to the built-in solver. Since we will usually have no information about the internal representation of B, we treat it as a conjunction of constraints. The *propagation history* T is a set of sequences, each recording the identities of the CHR constraints which fired a rule, and the name of the rule itself (which may be represented as a unique integer, but typically we just use the name of the rule itself). This is necessary to prevent trivial nontermination for propagation rules: a propagation rule is allowed to fire on a set of constraints only if the constraints have not been used to fire the rule before. The set $\mathcal{V}$ contains all variables that appeared in the initial goal. Throughout this paper we will usually omit $\mathcal{V}$ unless it is explicitly required. Finally, the counter n represents the next free integer which can be used to number a CHR constraint.

Throughout we let $vars(A)$ return the variables occurring in any syntactic object A. We use $\exists_A F$ to denote the formula $\exists X_1 \cdots \exists X_n F$ where $\{X_1, \ldots X_n\} = vars(A)$. We use $\bar{\exists}_A F$ to denote the formula $\exists X_1 \cdots \exists X_n F$ where $\{X_1, \ldots X_n\} = vars(F) - vars(A)$.

We use $[H]$ to construct a sequence of length 1 containing element H, $+\!\!+$ for sequence concatenation, and $\uplus$ for multiset union. We shall sometimes treat multisets as sequences, in which case we nondeterministically choose an order for the objects in the multiset. We use the notation $p(s_1, \ldots, s_n) = p(t_1, \ldots, t_n)$ as shorthand for the constraint $s_1 = t_1 \wedge \cdots \wedge s_n = t_n$, and similarly $S = T$ where S and T are equal length sequences $S \equiv s_1 \ \cdots \ s_n$ and $T = t_1 \ \cdots \ t_n$ as shorthand for $s_1 = t_1 \wedge \cdots \wedge s_n = t_n$.

We define an *initial state* as follows.

[4] Sometimes we treat the store as a multiset.

Definition 3 (Initial State). *Given a goal G, which is a multiset of constraints, the* initial state *with respect to G is $\langle G, \emptyset, true, \emptyset \rangle_1^{vars(G)}$.* □

The theoretical operational semantics ω_t is based on the following three transitions which map execution states to execution states:

Definition 4 (Theoretical Operational Semantics).
1. Solve

$$\langle \{c\} \uplus G, S, B, T \rangle_n^{\mathcal{V}} \rightarrowtail \langle G, S, c \wedge B, T \rangle_n^{\mathcal{V}}$$

where c is a built-in constraint.
2. Introduce

$$\langle \{c\} \uplus G, S, B, T \rangle_n^{\mathcal{V}} \rightarrowtail \langle G, \{c\#n\} \uplus S, B, T \rangle_{(n+1)}^{\mathcal{V}}$$

where c is a CHR constraint.

3. Apply

$$\langle G, H_1 \uplus H_2 \uplus S, B, T \rangle_n^{\mathcal{V}} \rightarrowtail \langle C \uplus G, H_1 \uplus S, B \wedge \theta, T' \rangle_n^{\mathcal{V}}$$

where there exists a (renamed apart) rule r in P of the form

$$H_1' \setminus H_2' \iff g \mid C$$

and $\theta \equiv chr(H_1) = H_1' \wedge chr(H_2) = H_2'$ such that

$$\begin{cases} \mathcal{D} \models B \rightarrow \exists_r (\theta \wedge g) \\ id(H_1) +\!\!+ id(H_2) +\!\!+ [r] \notin T \end{cases}$$

In the result $T' = T \cup \{id(H_1) +\!\!+ id(H_2) +\!\!+ [r]\}$. □

A *derivation* is a sequence of states connected by ω_t transitions. We use notation $\sigma_0 \rightarrowtail^* \sigma_1$ to represent a derivation from σ_0 to σ_1. We also define *complete derivation* to be a derivation from an initial state to a *final state* (i.e. a state where no transition is applicable).

2.1 Confluence

In this section we define confluence under the theoretical semantics ω_t.

First we define an auxiliary function *alive*, which decides which part of the propagation history is relevant for confluence.

Definition 5 (Live History). *Function alive is a bijective mapping from a CHR store S and a propagation history to a propagation history defined as follows.*

$$\begin{array}{lll} alive(S, \emptyset) & = \emptyset & \\ alive(S, \{t\} \uplus T) & = alive(S, t) \uplus alive(S, T) & \\ alive(S, t +\!\!+ [\_]) & = \emptyset & \text{if } \exists i \in t \text{ such that } \forall c(c\#i \notin S) \\ alive(\_, t) & = \{t\} & \text{otherwise} \end{array}$$

□

In other words, $alive(S,T)$ is propagation history T where all entries with numbers for deleted (i.e. not alive) constraints have been removed. Interestingly, $alive(S,T)$ can only have entries on propagation rules (otherwise one of the numbers in the entry must be dead).

Definition 6 (Variants). *Two states*

$$\sigma_1 = \langle G_1, S_1, B_1, T_1 \rangle^{\mathcal{V}}_{i_1} \qquad \textit{and} \qquad \sigma_2 = \langle G_2, S_2, B_2, T_2 \rangle^{\mathcal{V}}_{i_2}$$

(from either semantics) are variants *(i.e. $\sigma_1 \approx \sigma_2$) if there exists a renaming ρ on variables not in $\mathcal{V}$ and a mapping ϱ on constraint numbers such that*

1. $\rho \circ \varrho(G_1) = G_2$
2. $\rho \circ \varrho(S_1) = S_2$;
3. $\mathcal{D} \models (\bar{\exists}_{\mathcal{V}} \rho(B_1) \leftrightarrow \bar{\exists}_{\mathcal{V}} B_2)$; *and*
4. $\varrho \circ alive(S_1, T_1) = alive(S_2, T_2)$.

Otherwise the two states are variants if $\mathcal{D} \models \neg\bar{\exists}_{\emptyset} B_1$ and $\mathcal{D} \models \neg\bar{\exists}_{\emptyset} B_2$ (i.e. both states are $false$). □

Definition 7 (Joinable). *Two states σ_1 and σ_2 are* joinable *if there exists states σ_1' and σ_2' such that $\sigma_1 \rightarrowtail^* \sigma_1'$ and $\sigma_2 \rightarrowtail^* \sigma_2'$ and σ_1' and σ_2' are variants.* □

Definition 8 (Confluence). *A CHR program P is* confluent *if the following holds for all states σ_0, σ_1 and σ_2: If $\sigma_0 \rightarrowtail^* \sigma_1$ and $\sigma_0 \rightarrowtail^* \sigma_2$ then σ_1 and σ_2 are joinable.* □

2.2 Confluence Test

The confluence test for CHRs depends on calculating all critical pairs between rules in the program.

Definition 9 (Critical Pair). *Given two (renamed apart) rules r1 and r2 from program P of the respective forms*

$$H_{1r} \setminus H_{1k} \Longleftrightarrow g_1 \mid B_1.$$
$$H_{2r} \setminus H_{2k} \Longleftrightarrow g_2 \mid B_2.$$

Let $H_1 = H_{1r} \uplus H_{1k}$ and $H_2 = H_{2r} \uplus H_{2k}$. Let $H_1' \subseteq H_1$ and $H_2' \subseteq H_2$, and let θ be a most general unifier of multisets H_1' and H_2'. Given $\theta(H_1) \uplus (H_2 - H_2') = \{h_1, ..., h_m\}$ (for some arbitrary ordering), let $S = \{h_1\#1, ..., h_m\#m\}$. Also define $S_{1k} \subseteq S$ such that $chr(S_{1k}) = \theta(H_{1k})$ and $S_{2k} \subseteq S$ such that $chr(S_{2k}) = H_{2k}$. Let T_∞ be a propagation history such that for all propagation rules $r \in P$ we have that $[i_1, ..., i_k, r] \in T_\infty$ for all permutations of all subsets $\{i_1, ..., i_k\} \in \{1, .., n\}$. Then states

$$\sigma_1 = \langle B_1, S - S_{1k}, g_1 \wedge g_2, T_\infty \rangle^{\mathcal{V}}_n$$
$$\sigma_2 = \langle B_2, S - S_{2k}, g_1 \wedge g_2, T_\infty \rangle^{\mathcal{V}}_n$$

are a critical pair *between rules r1 and r2.* □

Definition 10 (Confluence Test). *Given a terminating CHR program P, if all critical pairs between all rules in P are joinable, then P is confluent. This is known as the* confluence test *for CHRs.* □

It has been shown that the confluence test decides confluence for terminating CHR programs [1]. This relies on the fact that there are finitely many critical pairs for a given program.

2.3 Observable Confluence

In this section we formally define observable confluence with respect to reachability.

Definition 11 (Reachability). *An execution state σ is* reachable *if there exists an initial state $\sigma_0 = \langle G, \emptyset, true, \emptyset \rangle_1$ such that there exists a derivation $\sigma_0 \rightarrowtail^* \sigma$.* □

Definition 12 (Observable Confluence). *A CHR program P is* observable confluent *if the following holds for all states σ_0, σ_1 and σ_2 where σ_0 is a reachable state: If $\sigma_0 \rightarrowtail_\omega^* \sigma_1$ and $\sigma_0 \rightarrowtail_\omega^* \sigma_2$ then σ_1 and σ_2 are joinable.* □

Notice the differences between Definition 12 and Definition 8: we now require that σ_0 is a *reachable* state.

3 Observable Confluence: Examples

Confluence implies observable confluence, so if a CHR program P passes Abdennahder's confluence test, then that program is also observable confluent. In the introduction, we have seen that there exists CHR programs that are observable confluent by Definition 12, yet are not confluent by Definition 8.

Example 4. Here is again the type class program from the introduction.

```
class F x y z | x -> y
instance F Int Bool Float
```

The corresponding CHR program, according to the translation given in [8], is

```
r1 @ f(int,bool,float)    <=> true.
r2 @ f(A,B1,C), f(A,B2,D) ==> B1 = B2.
r3 @ f(int,B,C)           ==> B = bool.
```

This program is not confluent, as we saw in Example 2. Here we illustrate this formally using the full theoretical operational semantics. Consider the following state applicable to rules `r1` and `r2`. σ:

$$\langle \emptyset, \{f(int, bool, float)\#1, f(int, B2, D)\#2\}, true, \{[1, r3], [2, r3]\}\rangle_3$$

The propagation history prevents rule `r3` from firing on either constraint.

The state σ has two distinct derivations:

$$\sigma \rightarrowtail^* \langle \emptyset, \{f(int, bool, D)\#2\}, B2 = bool, \{[1, 2, r2], [1, r3], [2, r3]\}\rangle_3$$
$$\sigma \rightarrowtail^* \langle \emptyset, \{f(int, B2, D)\#2\}, true, \{[1, r3], [2, r3]\}\rangle_3$$

These two final states are non-variant, therefore the program is not confluent.

We claim this program is observable confluent. The state σ is not reachable since the propagation history indicates that rule `r3` has fired on constraint $f(int, B2, T)\#2$. However, if that were the case then we would expect the built-in constraint $B2 = bool$ to appear in the built-in store. This is not the case therefore the state σ cannot be reachable. □

We formally define the class of CHR programs arising from Haskell type class declarations with functional dependencies.

Definition 13 (FD-CHR). *A CHR program P is said to be in the* FD-CHR *class of programs if it is of the form*

`r1 @ p(`$X_1, ..., X_d, X_{d+1}, ..., X_r, ...$`), p(`$X_1, ..., X_d, Y_{d+1}, ..., Y_r, ...$`) ==>`
$X_{d+1} = Y_{d+1}, ..., X_r = Y_r$.
`r2 @ p(`$f_1, ..., f_n$`) <=>` B .
`r3 @ p(`$f_1, ..., f_d, Y_1, ..., Y_r, ...$`) ==>` $Y_1 = f_{d+1}, ..., Y_r = f_r$.

where B is an arbitrary conjunction of built-in and CHR constraints and f_i are arbitrary terms such that $vars(f_{d+1}, ..., f_r) \subseteq vars(f_1, ..., f_d)$ We also require P to be terminating. Here the indices $1..d$ represent the domain *and indices $(d+1)..r$ represent the* range *of the functional dependency. Also note that r is allowed to be less than n.* □

4 Observable Confluence: Formal Result

In this section we present the main result that relates observable confluence to ordinary confluence. These are related through an *operational correspondence*.

Definition 14 (Operational Correspondence). *Let P and P' be a CHR programs. An* operational correspondence *is a function α mapping complete derivations in P to complete derivations in P', and the following conditions for all derivations*

$$D_1 = (\sigma_i \rightarrowtail^*_P \sigma_{1f})$$
$$D_2 = (\sigma_i \rightarrowtail^*_P \sigma_{2f})$$
$$\alpha(D_1) = (\sigma_{1i} \rightarrowtail^*_{P'} \sigma'_{1f})$$
$$\alpha(D_2) = (\sigma_{2i} \rightarrowtail^*_{P'} \sigma'_{2f})$$

are satisfied:

1. *σ_{1i} is an initial state;*
2. *$\sigma_{1i} = \sigma_{2i}$; and*
3. *If $\sigma_{1f} \not\approx \sigma_{2f}$ then $\sigma'_{1f} \not\approx \sigma'_{2f}$.*

□

In other words, an operational correspondence preserves initial states, and preserves non-variance on final states. We can use operational correspondence to show observable confluence.

Theorem 1. *Let P and P' be a CHR programs such that P' is confluent and there exists an operational correspondence α from P to P', then P is observable confluent.*

Proof. By contradiction, assume P is not observable confluent. Therefore there exists an initial state σ_i and two complete derivations

$$D_1 = (\sigma_i \rightarrowtail^*_P \sigma_{1f})$$
$$D_2 = (\sigma_i \rightarrowtail^*_P \sigma_{2f})$$

such that σ_{1f} and σ_{2f} are two non-variant final states. By the operational correspondence we have that

$$\alpha(D_1) = (\sigma'_i \rightarrowtail^*_{P'} \sigma'_{1f})$$
$$\alpha(D_2) = (\sigma'_i \rightarrowtail^*_{P'} \sigma'_{2f})$$

for some initial state σ'_i and non-variant final states σ'_{1f} and σ'_{2f}. This contradicts the confluence of P', since there exists an initial state σ'_i which can be reduced to non-variant final states σ'_{1f} or σ'_{2f}. □

Example 5. Consider the following CHR program P

```
p(X), p(Y) ==> X = Y.
p(1) <=> true.
p(X) ==> X = 1
```

and the CHR program P'

```
p(X) <=> X = 1.
```

Proposition 1. *There exists an operational correspondence α from P to P' as follows:*

$$\alpha(\sigma_0 \rightarrowtail^*_P \langle \emptyset, S, B, T \rangle^{\mathcal{V}}_n) = \sigma_0 \rightarrowtail^*_P \langle \emptyset, S, B, \emptyset \rangle^{\mathcal{V}}_n$$

Note that there may be multiple possible derivations satisfying the RHS. If this is the case then we simply choose one arbitrarily to find a suitable function α.

The program P' is trivially confluent. Assuming Proposition 1 holds, then by Theorem 1 program P is observable confluent. □

5 Observable Confluence: FD-CHR

Using Theorem 1 we can reduce the problem of deciding observable confluence to the problem of finding a suitable P', such that P' is confluent and there exists a operational correspondence from P to P'. In this section we examine the FD-CHR class of programs, and use Theorem 1 to formally show they are observable confluent.

First we define the target program P' as follows.

Definition 15. *Given a program $P \in$ FD-CHR, we define the* correspondence program $\mathcal{C}(P)$ *for P as*

r1 @ p($X_1, ..., X_d, X_{d+1}, ..., X_r, ...$), p($X_1, ..., X_d, Y_{d+1}, ..., Y_r, ...$) ==>
$X_{d+1} = Y_{d+1}, ..., X_r = Y_r$.
r2' @ p($f_1, ..., f_n$) ==> B.
r3 @ p($f_1, ..., f_d, Y_1, ..., Y_r, ...$) ==> $Y_1 = f_{d+1}, ..., Y_r = f_r$.

□

The only difference is between rules r2 and r2': r2 is a simplification rule and r2' is an equivalent propagation rule. The remaining structure of the program is preserved. Note that $\mathcal{C}(P)$ is terminating since P is terminating.[5]

5.1 $\mathcal{C}(P)$ Confluence

In this section we establish that the class of $\mathcal{C}(P)$ is confluent.

Lemma 1. *For all $P \in$ FD-CHR, $\mathcal{C}(P)$ is confluent.*

Proof. Direct proof. We apply the standard CHR confluence test. Since there are only propagation rules in $\mathcal{C}(P)$, all critical pairs are trivially joinable. Therefore $\mathcal{C}(P)$ is confluent. □

5.2 $\mathcal{C}(P)$ Operational Correspondence

In this section we establish an operational correspondence between P and $\mathcal{C}(P)$.

First we define two auxiliary tests to help make the proofs more concise.

Definition 16. *We define a test $\mathsf{Inst}_B(p(F_1, ...F_n))$ that succeeds if there exists a substitution θ such that*

$$D \models B \rightarrow (p(F_1, ...F_n) = p(\theta.f_1, ..., \theta.f_n))$$

Similarly we define $\mathsf{Inst}_B^{DOM}(p(F_1, ...F_n))$ that succeeds if there exists a substitution θ such that

$$D \models B \rightarrow (p(F_1, ...F_n) = p(\theta.f_1, ..., \theta.f_d, F_{d+1}, ..., F_n)))$$

□

Informally, Inst_B defines the set of all constraints that match r2 (or r2'), and Inst_B^{DOM} defines the set of all constraints that match r3.

Lemma 2. *If*

$$\sigma_0 \rightarrowtail_P^* \langle G, S, B, T\rangle_n$$

then

$$\sigma_0 \rightarrowtail_{\mathcal{C}(P)}^* \langle G, S \uplus S', B, T \cup T'\rangle_n$$

where S' is

$$\{p(F_1^1, ..., F_n^1)\#i_1, ..., p(F_1^m, ..., F_n^m)\#i_m\}$$

for some set of constraint numbers $i_1, ..., i_m$, $\mathsf{Inst}_B(p(F_1^j, ...F_n^j))$ holds for all $j \in 1, \ldots, m$. And for all $t \in T'$, t is of the form $[j, r2']$ for some $j \in i_1, ..., i_m$

[5] We omit a formal proof for space reasons.

Proof. By induction over the derivation steps in P.
Base case: No derivation steps. Then $S' = \emptyset$ and $T' = \emptyset$.
Induction step: Suppose for derivations D_i of length i of the form

$$\sigma_0 \rightarrowtail_P^* \langle G_i, S_i, B_i, T_i \rangle_{n_i}$$

we have that there exists a derivation D'_i of the form

$$\sigma_0 \rightarrowtail_{\mathcal{C}(P)}^* \langle G_i, S_i \uplus S'_i, B_i, T_i \cup T'_i \rangle_{n_i}$$

where S'_i and T'_i satisfy the conditions on S' and T' from above respectively.

Consider all derivations of length $i+1$ constructed from applying an ω_t transition to the final state in D_i. The possible transitions are:

- **Solve**. Then $G_i = \{c\} \uplus G'_i$ for some built-in constraint c. Thus

$$\langle \{c\} \uplus G'_i, S_i, B_i, T_i \rangle_{n_i} \rightarrowtail_{\mathbf{Solve}} \langle G'_i, S_i, c \wedge B_i, T_i \rangle_{n_i}$$

  and for $\mathcal{C}(P)$

$$\langle \{c\} \uplus G'_i, S_i \uplus S'_i, B_i, T_i \cup T'_i \rangle_{n_i} \rightarrowtail_{\mathbf{Solve}} \langle G'_i, S_i \uplus S'_i, c \wedge B_i, T_i \cup T'_i \rangle_{n_i}$$

  Thus the induction hypothesis holds for $i+1$ with $S'_{i+1} = S'_i$ and $T'_{i+1} = T'_i$.
- **Introduce**. Then $G_i = \{c\} \uplus G'_i$ for some CHR constraint c. Thus

$$\langle \{c\} \uplus G'_i, S_i, B_i, T_i \rangle_{n_i} \rightarrowtail_{\mathbf{Introduce}} \langle G'_i, \{c\#n_i\} \uplus S_i, B_i, T_i \rangle_{n_i+1}$$

  and for $\mathcal{C}(P)$

$$\langle \{c\} \uplus G'_i, S_i \uplus S'_i, B_i, T_i \cup T'_i \rangle_{n_i} \rightarrowtail_{\mathbf{Introduce}} \langle G'_i, \{c\#n_i\} \uplus S_i \uplus S'_i, B_i, T_i \cup T'_i \rangle_{n_i+1}$$

  Thus the induction hypothesis holds for $i+1$ with $S'_{i+1} = S'_i$ and $T'_{i+1} = T'_i$.
- **Apply**. We split this case into two smaller cases:
  1. **Apply** r2. Then $S_i = \{p(F_1, ..., F_n)\#j\} \uplus S$ for some constraint number j and CHR store S where $\mathsf{Inst}_{B_i}(p(F_1, ..., F_n))$. Thus

$$\langle G_i, \{p(F_1, ..., F_n)\#j\} \uplus S, B_i, T_i \rangle_{n_i} \rightarrowtail_{\mathbf{Apply}} \langle B \wedge G_i, S, B_i \wedge \theta, T_i \rangle_{n_i}$$

     and

$$\langle G_i, \{p(F_1, ..., F_n)\#j\} \uplus S \uplus S'_i, B_i, T_i \cup T'_i \rangle_{n_i} \rightarrowtail_{\mathbf{Apply}} \langle B \wedge G_i, \{p(F_1, ..., F_n)\#j\} \uplus S \uplus S'_i, B_i \wedge \theta, T_i \cup T'_i \cup \{[j, r2']\} \rangle_{n_i}$$

     Thus the induction hypothesis holds for $i+1$ with $S'_{i+1} = S'_i \uplus \{p(F_1, ..., F_n)\#j\}$ and $T'_{i+1} = T'_i \cup \{[j, r2']\}$.
  2. **Apply** r1 $\vee$ r3. Either r1 or r3 is applicable to the constraints in S_i. Thus

$$\langle G_i, S_i, B_i, T_i \rangle_{n_i} \rightarrowtail_{\mathbf{Apply}} \langle C \uplus G_i, S_i, B_i \wedge \theta, \{t\} \cup T_i \rangle_{n_i}$$

     and using $\mathcal{C}(P)$

$$\langle G_i, S_i \uplus S'_i, T_i \cup T'_i \rangle_{n_i} \rightarrowtail_{\mathbf{Apply}} \langle C \uplus G_i, S_i \uplus S'_i, B_i \wedge \theta, \{t\} \cup T_i \cup T'_i \rangle_{n_i}$$

     for some θ, C and t. Thus the induction hypothesis holds for $i+1$ with $S'_{i+1} = S'_i$ and $T'_{i+1} = T'_i$.

□

Lemma 3. *Given a complete derivation under P*

$$\sigma_0 \rightarrowtail_P^* \langle \emptyset, S, B, T\rangle_n^{\mathcal{V}} = \sigma_f$$

and the corresponding derivation under $\mathcal{C}(P)$ (given by Lemma 2)

$$\sigma_0 \rightarrowtail_{\mathcal{C}(P)}^* \langle \emptyset, S \cup S', B, T \cup T'\rangle_n^{\mathcal{V}} = \sigma_1$$

suppose that $\sigma_1 \rightarrowtail_{\mathcal{C}(P)}^* \sigma_i$ then σ_i is of the form

$$\langle G_i, S \cup S', B_i, T \cup T' \cup T_i\rangle_n^{\mathcal{V}}$$

such that

1. *$\mathcal{D} \models \bar{\exists}_{\mathcal{V}} B \leftrightarrow \bar{\exists}_{\mathcal{V}} B_i$;*
2. *for all $c \in G_i$, c is built-in and $\mathcal{D} \models B_i \rightarrow c$;*
3. *for all $t \in T_i$ we have that $id(t) \cap id(S') \neq \emptyset$.*

Proof. By induction over the derivation steps in $\sigma_1 \rightarrowtail_{\mathcal{C}(P)}^* \sigma_i$
Base case: $i = 1$, thus $G_i = \emptyset$, $B_i = B$ and $T_i = \emptyset$ satisfies the conditions.
Induction step: Suppose that for $i - 1$ we have that

$$\sigma_{i-1} = \langle G_{i-1}, S \cup S', B_{i-1}, T \cup T' \cup T_{i-1}\rangle_n^{\mathcal{V}}$$

is of the required form.

We consider all ω_t transition steps applicable to σ_{i-1}.

- **Solve**. Then $G_{i-1} = c \wedge G'_{i-1}$ for some built-in constraint c, thus

  $$\sigma_{i-1} \rightarrowtail_{\mathbf{Solve}} \langle G'_{i-1}, S \cup S', c \wedge B_{i-1}, T \cup T' \cup T_{i-1}\rangle_n^{\mathcal{V}}$$

  We see that $G_i = G'_{i-1} \subset G_{i-1}$ and $T_i = T_{i-1}$ satisfies conditions (2) and (3) respectively. For condition (1): Since $\mathcal{D} \models \bar{\exists}_{\mathcal{V}} B \leftrightarrow \bar{\exists}_{\mathcal{V}} B_{i-1}$ and $\mathcal{D} \models B \rightarrow c$ we have that $B_i = (c \wedge B_{i-1})$ satisfies $\mathcal{D} \models \bar{\exists}_{\mathcal{V}} B \leftrightarrow \bar{\exists}_{\mathcal{V}} B_i$. Thus condition (1) is satisfied.
- **Introduce**. This transition is not applicable since there are no CHR constraints in G_{i-1}.
- **Apply**. A rule is applied to the constraints in $S \uplus S'$. Thus

  $$\sigma_{i-1} \rightarrowtail_{\mathbf{Apply}} \langle C \uplus G_{i-1}, S \cup S', B_{i-1} \wedge \theta, T \cup T_{i-1} \cup \{t\}\rangle_n$$

  for some entry t, rule body C and matching substitution θ.
  Now $D \models B \rightarrow \exists_r \theta$ and $D \models \bar{\exists}_{\mathcal{V}} B \leftrightarrow \bar{\exists}_{\mathcal{V}} B_{i-1}$ hence $D \models \bar{\exists}_{\mathcal{V}} B \leftrightarrow \bar{\exists}_{\mathcal{V}} (B_{i-1} \wedge \theta)$ satisfying condition (1). Let $M \subseteq S \uplus S'$ be the *matching*, then $M \cap S' \neq \emptyset$ otherwise σ_f is not a final state. Thus $id(t) \cap id(S') \neq \emptyset$ and hence $T_i = T_{i-1} \cup \{t\}$ satisfies condition (3).
  Next consider $G_i = C \uplus G_{i-1}$. For condition (2) to hold, we need to show that for all $c \in C$ we have that c is a built-in constraint, and $\mathcal{D} \models B_{i-1} \wedge \theta \rightarrow c$. There are three cases to consider:

1. **Apply** r1. Then $M = \{p(F_1, ..., F_n)\#j, p(T_1, ..., T_n)\#k\}$, for some constraint numbers j, k. C is of the form:

$$\{X_{d+1} = Y_{d+1}, ..., X_r = Y_r\} \tag{1}$$

Obviously C is all built-in constraints as required. Also, $\theta \equiv (\bigwedge_{l=1}^{n} F_l = X_l) \wedge (\bigwedge_{l=1}^{d} T_l = X_l) \wedge (\bigwedge_{l=d+1}^{n} T_l = Y_l)$ and $D \models B_{i-1} \rightarrow \exists \bar{X} \exists \bar{Y} \theta$.
Since $M \cap S' \neq \emptyset$ we can assume w.l.o.g. $p(F_1, ..., F_n)\#k \in S'$. Therefore $\mathsf{Inst}_{B_{i-1}}(p(F_1, ..., F_n))$ holds.
From $D \models B_{i-1} \rightarrow \exists \bar{X} \exists \bar{Y} \theta$ we project out $\bar{X}$ and $\bar{Y}$ to derive

$$D \models B_{i-1} \rightarrow \bigwedge_{l=1}^{d} F_l = T_l \tag{2}$$

I.e. the domain arguments must coincide. Therefore $\mathsf{Inst}^{DOM}_{B_{i-1}}(p(T_1, ..., T_n))$ must hold, and r3 is applicable to this rule. Since σ_f is a final state, rule r3 must have already been applied to this constraint, and thus we can conclude $\mathsf{Inst}_{B_{i-1}}(p(T_1, ..., T_n))$ holds.
Since $vars(f_{d+1}, ..., f_r) \subseteq vars(f_1, ..., f_d)$, and by (2) we have that

$$D \models B_{i-1} \rightarrow \bigwedge_{l=d+1}^{r} F_l = T_l$$

Therefore

$$D \models (B_{i-1} \wedge \theta) \rightarrow \bigwedge_{l=d+1}^{r} X_l = Y_l$$

I.e. the equations of C are already implied by $B_i = (B_{i-1} \wedge \theta)$ and thus condition (2) is satisfied.

2. **Apply** r2'. Then $M = \{p(F_1, ..., F_n)\#j\} \subseteq S'$ (since $M \cap S' \neq \emptyset$) for some constraint number j,
Since $M \cap S' \neq \emptyset$ then by Lemma 2 there exists an entry $[j, r2'] \in T'$, thus **Apply** is not applicable so we can exclude this case.
3. **Apply** r3. Then $M = \{p(F_1, ..., F_n)\#j\} \subseteq S'$ (since $M \cap S' \neq \emptyset$) for some constraint number j, and hence $\mathsf{Inst}_{B_{i-1}}(p(F_1, ..., F_n))$ holds.
Now $\theta \equiv (\bigwedge_{l=1}^{d} F_l = f_l) \wedge (\bigwedge_{l=d+1}^{n} F_l = Y_l)$ and C is of the form

$$\{Y_{d+1} = f_{d+1}, ..., Y_r = f_r\}$$

Since $vars(f_{d+1}, ..., f_r) \subseteq vars(f_1, ..., f_d)$, we have that clearly $\mathcal{D} \models (B_{i-1} \wedge \theta) \rightarrow c$ for all $c \in C$ and thus condition (2) holds.

In either case the state σ_i satisfies the required conditions.

□

Lemma 4. *There exists an operational correspondence between $P \in$ FD-CHR and $\mathcal{C}(P)$.*

Proof. Direct proof. By Lemma 3 and termination of $\mathcal{C}(P)$, for a complete derivation D in P

$$\sigma_0 \rightarrowtail_P^* \sigma_f = \langle \emptyset, S, B, T \rangle_n^{\mathcal{V}}$$

there exists a complete derivation D' in $\mathcal{C}(P)$

$$\sigma_0 \rightarrowtail_{\mathcal{C}(P)}^* = \langle \emptyset, S \uplus S', B_i, T \cup T' \cup T_i \rangle_n^{\mathcal{V}}$$

where S', B_i, T' and T_i satisfy the conditions of Lemmas 2 and 3.

Define $\alpha(D) = D'$. Next we check that α satisfies the conditions for Definition 14. Given the complete derivations

$$\begin{aligned} D_1 &= (\sigma_0 \rightarrowtail_P^* \sigma_{1f}) \\ D_2 &= (\sigma_0 \rightarrowtail_P^* \sigma_{2f}) \\ \alpha(D_1) &= (\sigma_{1i} \rightarrowtail_{P'}^* \sigma'_{1f}) \\ \alpha(D_2) &= (\sigma_{2i} \rightarrowtail_{P'}^* \sigma'_{2f}) \end{aligned}$$

we have that

1. σ_{1i} is an initial state since $\sigma_{1i} = \sigma_0$;
2. $\sigma_{1i} = \sigma_{2i}$ since also $\sigma_{2i} = \sigma_0$; and
3. if $\sigma_{1f} \not\approx \sigma_{2f}$ then $\sigma'_{1f} \not\approx \sigma'_{2f}$. We show this condition is satisfied by contradiction. Assume that $\sigma_{1f} \not\approx \sigma_{2f}$ but $\sigma'_{1f} \approx \sigma'_{2f}$.
   Let

$$\begin{aligned} \sigma_{1f} &= \langle \emptyset, S_1, B_1, T_1 \rangle_n^{\mathcal{V}} \\ \sigma_{2f} &= \langle \emptyset, S_2, B_2, T_2 \rangle_m^{\mathcal{V}} \\ \sigma'_{1f} &= \langle \emptyset, S_1 \uplus S'_1, B_{1i}, T_1 \cup T'_1 \cup T_{1i} \rangle_n^{\mathcal{V}} \\ \sigma'_{2f} &= \langle \emptyset, S_2 \uplus S'_2, B_{2i}, T_2 \cup T'_2 \cup T_{2i} \rangle_m^{\mathcal{V}} \end{aligned}$$

   where S'_1, S'_2, B_{1i}, B_{2i}, T'_1, T'_2, T_{1i} and T_{2i} satisfy the conditions of Lemmas 2 and 3 in the obvious manner.
   Given Definition 6, for all renamings ρ on variables not in $\mathcal{V}$ and mappings ϱ on constraint numbers, there are four cases the consider:
   (a) *Goals:* $\rho \circ \varrho(\emptyset) \neq \emptyset$ gives an immediate contradiction.
   (b) *CHR Stores:* Given that $\rho \circ \varrho(S_1) \neq S_2$ then, w.l.o.g., there exists a numbered constraint $c\#j \in \rho \circ \varrho(S_1)$ such that $c\#j \notin S_2$.
   Thus if $\rho \circ \varrho(S_1 \uplus S'_1) = (S_2 \uplus S'_2)$ we have that $c\#j \in S_2 \uplus S'_2$, therefore $c\#j \in S'_2$. Thus c must satisfy $\mathsf{Inst}_{B_{2i}}(c)$, and hence rule r2 is applicable to $c\#j$ in state σ_{1f}. Therefore σ_{1f} cannot be a final state which is a contradiction.
   (c) *Built-in Stores:* Given that $\mathcal{D} \models (\bar{\exists}_{\mathcal{V}} \rho(B_1) \not\leftrightarrow \bar{\exists}_{\mathcal{V}} B_2)$ assume that $\mathcal{D} \models (\bar{\exists}_{\mathcal{V}} \rho(B_{1i}) \leftrightarrow \bar{\exists}_{\mathcal{V}} B_{2i})$.
   We immediately arrive at a contradiction since $\mathcal{D} \models \bar{\exists}_{\mathcal{V}} B_1 \leftrightarrow \bar{\exists}_{\mathcal{V}} B_{1i}$ and $\mathcal{D} \models \bar{\exists}_{\mathcal{V}} B_2 \leftrightarrow \bar{\exists}_{\mathcal{V}} B_{2i}$.

(d) *Histories:* Let $T_{1a} = \varrho \circ alive(S_1, T_1)$ and $T_{2a} = alive(S_2, T_2)$. Given that $T_{1a} \neq T_{2a}$, w.l.o.g., there exists a propagation history entry t such that $t \in T_{1a}$ but $t \notin T_{2a}$.
Let

$$T'_{1a} = \varrho \circ alive(S_1 \uplus S'_1, T_1 \uplus T'_1 \uplus T_{1i})$$
$$T'_{2a} = alive(S_2 \uplus S'_2, T_2 \uplus T'_2 \uplus T_{2i})$$

By assumption, $T'_{1a} = T'_{2a}$. By Definition 5 we see that $T_{1a} \subseteq T'_{1a}$, thus $t \in T'_{1a}$. Since $T'_{1a} = T'_{2a}$ we have that $t \in T'_{2a}$.
Expanding out the shorthand notation, the above is equivalent to

$$t \in \varrho \circ alive(S_1, T_1)$$
$$t \notin alive(S_2, T_2)$$
$$t \in alive(S_2 \uplus S'_2, T_2 \uplus T'_2 \uplus T_{2i})$$

There are three cases to consider:

i. $t \in T'_2$. Then $t = [j, r2']$ for some j. We immediately arrive at a contradiction since rule r2' is not present in program P, yet the propagation history for σ_{1f} (namely T_1) mentions it.

ii. $t \in T'_{2i}$. Then by Lemma 3 we have that $id(t) \cap id(S'_2) \neq \emptyset$. Thus there exists a $c\#j \in S'_2$ such that $j \in t$. Therefore $\mathsf{Inst}_{B_{2i}}(c)$ holds. Since $t \in \varrho \circ alive(S_1, T_1)$ we have that $c\#j' \in S_1$ where $\varrho(j') = j$. Since $\mathcal{D} \models \bar{\exists}_{\mathcal{V}} B_{1i} \leftrightarrow \bar{\exists}_{\mathcal{V}} B_{2i}$ we have that $\mathsf{Inst}_{B_{1i}}(c)$ also holds, thus constraint c is applicable to rule r2, therefore σ_{1f} is not a final state. This is a contradiction.

iii. $t \in T_2$. If $id(t) \subset id(S_2)$ we instantly reach a contradiction, since this implies $t \in alive(S_2, T_2)$. Therefore $id(t) \cap id(S'_2) \neq \emptyset$, however this leads to the same contradiction as preceding case.

Each case leads to a contradiction, therefore if $\sigma_{1f} \not\approx \sigma_{2f}$ then $\sigma'_{1f} \not\approx \sigma'_{2f}$.

Therefore α is an operational correspondence between P and $\mathcal{C}(P)$. □

5.3 Main result

Corollary 1. *All programs $P \in$ FD-CHR are observable confluent.*

Proof. Directly follows from Theorem 1 and Lemmas 1 and 4. □

6 Conclusion and Future Work

We have investigated observable confluence for a class of non-confluent CHRs which arise when building type inferencer for type class programs with functional dependencies [8]. Our results guarantee that all reachable states during the type class inference process are confluent. Thus, we obtain completeness of inference for a larger set of type class programs. For simplicity we considered the cases arising from one instance and one functional dependency, the results extend straightforwardly for program arising from many instance declarations

and functional dependencies (assuming the instances satisfy the functional dependencies).

In future work, we plan to consider further classes of CHRs which are observable confluent. There are a number of other situations where weaker notions of observable confluence are also important. We briefly sketch one of these cases.

Lam and Sulzmann [6] are using CHRs for agent-oriented programming. These CHRs fail the confluence test. However, in practice CHRs are only applied to constraint stores which satisfy certain invariants of the agent world. CHR applications maintain these invariants. Hence, we should obtain observable confluence for all initial states which satisfy a certain condition. For example, consider the following two simplified CHRs modelling the block world, a standard example of an agent world.

`g1 @ get(X), empty   <=>` rhs_1
`g2 @ get(X), hold(Y) <=>` rhs_2

The critical pair `get(X), empty, hold(Y)` is not be joinable depending on the right-hand sides. Hence, the above CHRs are non-confluent. However, we know that the CHRs for the block world obey the invariant that `empty` and `hold(Y)` will never appear in any constraint store at the same time. Hence, we argue that the CHRs are observable confluent with respect to the condition that initial constraint store does not contain `empty` and `hold(Y)`.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proc. of CP'97*, LNCS, pages 252–266. Springer-Verlag, 1997.
2. G.J. Duck, M. García de la Banda, P.J. Stuckey, and C. Holzbaur. The refined operational semantics for constraint handling rules. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, LNCS, pages 120–136. Springer-Verlag, 2004.
3. T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.
4. P. J. Stuckey G. J. Duck, S. Peyton-Jones and M. Sulzmann. Sound and decidable type inference for functional dependencies. In *Proc. of ESOP'04*, volume 2986 of *LNCS*, pages 49–63. Springer-Verlag, 2004.
5. M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.
6. E. S. L. Lam and M. Sulzmann. Representing linear logic agents in CHR. http://www.comp.nus.edu.sg/~ sulzmann, May 2006.
7. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
8. M. Sulzmann, G.J. Duck, S. Peyton-Jones, and P.J. Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, page to appear, 2006.

Complexity of the CHR Rational Tree Equation Solver

Marc Meister and Thom Frühwirth

Fakultät für Informatik, Universität Ulm, Germany
{Marc.Meister, Thom.Fruehwirth}@uni-ulm.de

Abstract. Constraint Handling Rules (CHR) is a concurrent, committed-choice, rule-based language. One of the first CHR programs is the classic constraint solver for syntactic equality of rational trees that performs unification [7, 4, 14]. The worst-case time (and space) complexity of this short and elegant solver so far was an open problem [8] and assumed to be *polynomial*. In this paper we show that under the standard operational semantics of CHR there exist particular computations with n occurrences of variables and function symbols that produce $O(2^n)$ constraints, thus leading to *exponential* time and space complexity. We also show that the standard implementation of the solver in CHR libraries for Prolog may not terminate due to the Prolog built-in order used in comparing terms. Complexity can be improved to be *quadratic for any term order* under both standard and refined CHR semantics without changing the equation solver, when equations are transformed into flat normal form.

1 Introduction

Unification Algorithms. Unification is concerned with making first order logic terms syntactically equivalent by substituting terms for variables. For example, the terms $h(a, f(Y))$ and $h(Y, f(a))$ can be made identical by substituting the variable Y by a. In 1930, Herbrand [9] gave an informal description of a unification algorithm. Robinson [13] rediscovered a similar algorithm when he introduced the resolution procedure for first-order logic in 1965. Resolution and unification form the computational basis of logic programming languages such as Prolog. Since the late 70s, there are quasi-linear time algorithms for unification. For finite trees (Herbrand terms), see [11] and [12]. For rational trees, see [10]. These algorithms can be considered as extensions of the union-find algorithm [17] from constants to trees.

Syntactic Equations. In constraint programming, unification of terms is understood as solving equations, e.g., the equation $h(a, f(Y))$ `eq` $h(Y, f(a))$ will reduce to the solved normal form Y `eq` a. Syntactic equality is an essential ingredient of constraint logic programming, since terms are the universal data structure and equalities can be used to build, access, and take apart terms. In early Prolog implementations, the *occur-check* was omitted from syntactic equality for efficiency reasons. The result was that equation solving could go into an

infinite loop (e.g. for X `eq` $f(X)$). In Prolog II, an algorithm for properly handling the resulting infinite terms was introduced [3]. This class of infinite terms is called rational trees and is introduced in Section 2.

CHR Rational Tree Solver. Constraint Handling Rules CHR [4, 8, 14] is a concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multi-sets of constraints (atomic formulas) into simpler ones until they are solved. Like other algorithms for rational tree unification, the CHR rational tree solver (c.f. Section 3, Fig. 1) relies on a size-based order on terms to ensure termination. However, a formal termination proof for the solver is not available so far[1]. Standard proof methods and counter-examples for unification algorithms do not apply, since the CHR solver does not rely on solved variables and their substitutions. Even worse, the standard implementation of the solver uses the Prolog term order. We show that the solver does not terminate with that particular order.

Exponential Complexity. In Section 4, we investigate termination and worst-case time and space complexity of the solver when using a certain *measure order*. It is based on a measure that maps terms and constraints to natural numbers. To the best of our knowledge, this yields the first termination proof for an unification algorithm where a scalar suffices (usually lexicographic or multi-set orderings are used). Our main result is that there are computations under the standard CHR operational semantics for a problem with size n that require $O(2^n)$ rule applications in the worst-case. This exponential complexity is shown to be tight. To this end, we give a witness query with size $O(n^2)$ that produces more than 2^n constraints. Therefore, worst-case space and (hence) time complexity of the classic CHR constraint solver for unification is *exponential*. However, it is still an open problem, if the result carries over to actual implementations of the solver that usually rely on the refined CHR semantics.

Quadratic Complexity. But we also have good news in Section 5: The very same solver runs in quadratic time and space, as we prove, by requiring that the equations to be solved are in flat normal form. Such equations do not contain terms with nested applications of function symbols. For example, the equation $h(a, f(Y))$ `eq` $h(Y, f(a))$ can be flattened into $h(A, Z)$ `eq` $h(Y, X) \wedge A$ `eq` $a \wedge Z$ `eq` $f(Y) \wedge X$ `eq` $f(A)$. Since any set of syntactic equations can be transformed into flat normal form in linear time and space, this requirement is no real restriction. We also show that the results are rather independent of the term order used in the solver.

2 Rational Trees

A *rational tree* has a finite representation as a directed (possibly cyclic) graph by merging all nodes with common subtrees.

Definition 1. *A* rational tree *(or RT) is a (possibly infinite) finitely branching tree which has a finite number of subtrees.*

[1] This may be an example were elegance does not pay off: The solver consists of just four rules, and so is more concise than most formal specifications of unification.

A rational tree can also be represented as a binary equality constraint. For example, the infinite tree $f(f(f(\ldots)))$ only contains itself and can be represented by the equation X `eq` $f(X)$. (Variables are written in upper-case and function symbols in lower-case letters as common in logic programming.)

A conjunction of atomic constraints is *solved* (or in *solved normal form*) if it is either *false* or if it is of the form $\bigwedge_{i=1}^{n} X_i$ `eq` T_i with pairwise distinct variables $X_1, \ldots, X_n$ and arbitrary terms $T_1, \ldots, T_n$ for $n \in \mathbf{N}$. We require X_i to be different to T_j for $1 \leq i \leq j \leq n$, i.e., if a variable occurs on the l.h.s. of an equation, it does neither occur as the l.h.s. nor r.h.s. of any subsequent equation. The empty conjunction ($n = 0$) is identified with *true*. For example, the equations $f(X, b)$ `eq` $f(a, Y)$, X `eq` $t \wedge X$ `eq` s, and X `eq` $Y \wedge Y$ `eq` X are all *not in solved form*, while X `eq` $Z \wedge Y$ `eq` $Z \wedge Z$ `eq` t is *in solved form*. The solved form is not unique, e.g., X `eq` Y and Y `eq` X are logically equivalent but syntactically different solved forms, as are X `eq` $f(X)$ and X `eq` $f(f(X))$.

3 Rational Tree Equation Solver

The CHR program in Fig. 1 solves rational tree equations [7, 4, 8]. This solver dates back to late 1993 and was revised in 1998 [14]. The underlying algorithm is similar to the one in [3], but unlike this and most other unification algorithms it uses variable elimination (substitution) only in a very limited way, if it cannot be avoided. As a consequence, the algorithm has to rely on an order on variables. However, this simplification of the algorithm makes termination and complexity analysis considerably harder.

The auxiliary built-ins of the solver allows one to be independent of the representation of terms in the implementation: Besides *true* and *false*, we have var(T) iff T is a variable and fun(T) iff T is a function term (i.e. not a variable). A generic total order is implemented by $\prec$ and $\preceq$ and explained below in Subsection 3.2. The built-in `same_functor(T1,T2)` tests if `T1` and `T2` have the same function symbol and the same arity. It leads to *false* if not. The auxiliary CHR constraint `same_args(T1,T2)` pairwise equates the arguments of the two terms.

The rule `reflexivity` removes trivial equations between identical variables. The rule `orientation` reverses the arguments of an equation so that the (smaller) variable comes first. The order check in the guard makes sure that it is applicable only once to a given equation. The rule `decomposition` applies to function terms. When there is a clash, `same_functor` will lead to *false*. Otherwise, the initial equation between two function terms will be replaced by equations between the corresponding arguments of the terms. The rule `confrontation` replaces the variable in the second equation by the value of that variable according to the first equation. It performs a limited amount of variable elimination (substitution) by only considering the l.h.s.' of equations. This rule duplicates the term `T1`. For termination it is ensured by the guard that `T1` is not larger than `T2`.

Due to the `confrontation` rule, the complexity of the solver is worse than linear. The intricate interaction between the `decomposition` rule and the `con-`

`frontation` rule in the case of infinite terms (cyclic terms) makes it hard to determine the worst-case time complexity of the solver.

```
reflexivity   @ X eq X            <=> var(X) | true.
orientation   @ T eq X            <=> var(X), X≺T | X eq T.
decomposition @ T1 eq T2          <=> fun(T1), fun(T2) |
                                      same_functor(T1,T2), same_args(T1,T2).
confrontation @ X eq T1, X eq T2 <=> var(X), X≺T1, T1⪯T2 |
                                      X eq T1, T1 eq T2.
```

Fig. 1. Rational tree equation solver (RT solver)

The solver is satisfaction-complete, i.e. detects unsatisfiability: The conditions for the solved normal form can be restated as $X_i \prec X_{i+1}$ and $X_i \prec T_{i+1}$ (for $1 \leq i < n$) since any strict total order is transitive and asymmetric. Actually, the solver computes the solved form, as can be shown by contradiction: As long as a conjunction of constraints is not in solved form, at least one rule is applicable. If it is in solved form, no rule is applicable.

Example 1. Here is a simple example involving infinite rational trees that shows that one of the equations is redundant.

$$
\begin{aligned}
& \underline{\texttt{X eq f(X)}}\texttt{, } \underline{\texttt{X eq f(f(X))}} \\
\mapsto_{\texttt{confrontation}} \quad & \texttt{X eq f(X), } \underline{\texttt{f(X) eq f(f(X))}} \\
\mapsto_{\texttt{decomposition}}\mapsto^* \quad & \underline{\texttt{X eq f(X)}}\texttt{, } \underline{\texttt{X eq f(X)}} \\
\mapsto_{\texttt{confrontation}} \quad & \texttt{X eq f(X), } \underline{\texttt{f(X) eq f(X)}} \\
\mapsto_{\texttt{decomposition}}\mapsto^* \quad & \texttt{X eq f(X), } \underline{\texttt{X eq X}} \\
\mapsto_{\texttt{reflexivity}} \quad & \texttt{X eq f(X)}
\end{aligned}
$$

3.1 Term Size and Problem Size

We first define term and problem size and then the generic order $\prec$ based on term size that is used in the RT solver to compare terms.

Definition 2. *The* term size $\#T$ *of a term* T *is the number of occurrences of variables and function symbols. For two function terms* S *and* T*, we define the* term-size order $S \prec_s T$ *iff* $\#S < \#T$*. The* problem size $\#C$ *of a conjunction* $C = \bigwedge_{i=1}^{n} S_i \texttt{ eq } T_i$ *of equations with* $n \in \mathbf{N}$ *is defined as* $\#\left(\bigwedge_{i=1}^{n} S_i \texttt{ eq } T_i\right) := \sum_{i=1}^{n} \#S_i + \#T_i$.

For example, the problem size of (the empty conjunction) *true* is 0, the size of $X \texttt{ eq } f(a)$ is 3, and $X \texttt{ eq } f(b) \wedge f(b) \texttt{ eq } h(a)$ has size 7.

3.2 Generic Term Order and Termination

Ordering terms according to the number of occurrences of symbols is common in the rational tree literature.

Definition 3. *Any instance of the* generic strict total term order $\prec$ *must have the following three properties [8]:*

1. *For different variables X and Y, either $X \prec Y$ or $Y \prec X$.*
2. *Any variable is strictly smaller than any function term.*
3. *Function terms of smaller term size are also smaller in the order (term-size property).*

A function term *is a term that is not a variable. Two terms T_1 and T_2 are* equivalent w.r.t. $\prec$ *if neither $T_1 \prec T_2$ nor $T_2 \prec T_1$. Clearly, terms of same size may be syntactically different terms.*

Termination of the RT solver crucially relies on the generic order $\prec$. The rules `reflexivity` and `orientation` are applicable at most once to an equation. Application of `decomposition` produces equations between the arguments of the functions of the initial equations. Thus, the new equations have arguments of smaller size. Application of `confrontation` replaces one occurrence of X by T_1. The guard ensures that `X` $\prec T_1 \preceq T_2$. Therefore, as long as T_1 is a variable, it gets closer from below to T_2 but can never exceed it. If T_1 is a function term, then so must be T_2, and then only `decomposition` is applicable to the new equation T_1 `eq` T_2 produced by `confrontation`. The resulting equations, including the unchanged X `eq` T_1, will only contain terms that are strictly smaller than T_2. Since there is only a finite number of variables and sub-terms in a given problem and since the generic term order is thus well-founded, the solver terminates[2].

3.3 Non-Termination with Standard Prolog Order

The standard implementation of the RT solver [8, 14] uses the built-in Prolog order `@<`. Variables are identified by the built-in `var/1` and function terms by the built-in `nonvar/1`. The Prolog order `@<` compares arguments of function terms lexicographically from left to right, e.g., $f(Y, f(a, X))$ `@<` $f(a, X)$ but $f(a, X) \prec_s f(Y, f(a, X))$. The order `@<` therefore does not respect the term-size property, it is *not* an instance of the generic term order. As we show in the following example, this can cause non-termination for infinite rational trees:

Example 2. The query `X eq f(Y,f(a,X)), X eq f(a,X)` does not terminate.

$$\begin{array}{ll} & \underline{\texttt{X eq f(Y,f(a,X))}}\texttt{, }\underline{\texttt{X eq f(a,X)}} \\ \mapsto_{\texttt{confrontation}} & \texttt{X eq f(Y,f(a,X)), }\underline{\texttt{f(Y,f(a,X)) eq f(a,X)}} \\ \mapsto_{\texttt{decomposition}}\mapsto^* & \texttt{X eq f(Y,f(a,X)), Y eq a, }\underline{\texttt{f(a,X) eq X}} \\ \mapsto_{\texttt{orientation}} & \texttt{X eq f(Y,f(a,X)), Y eq a, X eq f(a,X)} \end{array}$$

Similarly, and containing only one binary function symbol, the computation for the query `X eq f(Y,f(f(X,Y),X)), X eq f(f(X,Y),X)` does not terminate. Note that in the next section we give another order that is also incompatible with the generic term order, but makes the RT solver provably terminate.

[2] A formal termination proof for the solver with generic term order is still missing.

4 Exponential Complexity

In this section we show that there exists a term order for the RT solver such that the worst-case time and space complexity of the solver can be exponential in the size of the problem. This term order, however, is not an instance of the generic term order commonly used in the solver.

We define a *problem measure*, which maps CHR constraints into natural numbers. It is based on a term measure that is exponential in the depth of the term. In the RT solver, we replace the generic order $\prec$ by the so-called *measure order* which is defined in terms of the measure.

We show for each rule that the problem measure of the head is always strictly greater than the problem measure of the body, provided the guard holds. In this way we not only formally prove termination, but also show that the problem measure gives us an upper bound on the number of rule applications (derivation length) [5]. Since the cost of a rule application can be made constant in the RT solver, the derivation length directly gives us the desired complexity result. A worst-case example then shows that the bound is actually tight.

4.1 Term Measure and Problem Measure

We give an inductive definition of our *term measure*[3]. As the RT solver does not introduce *new* variables, the number of different variables v of a problem is clearly bounded by its size. Hence, we can assume that all variables are elements of $\{X_1, \ldots, X_v\}$. The natural number v is called the *number of variables* of the problem.

Definition 4. *The* term measure $|T|$ *of a term* T *is defined as follows:*

$$|T| := \begin{cases} i & \text{if } T = X_i \\ n + 2\sum_{i=1}^{n} |T_i| & \text{if } T = f(T_1, \ldots, T_n) \end{cases}$$

Note that a constant (i.e., a null-ary function) has measure 0. Due to the factor 2 in the recursive definition, the measure of a term *can be exponential* in its size, consider, e.g., the term `f(f(f(a)))`.

We extend our term measure to equations and queries.

Definition 5. *For an equation* S `eq` T *with terms* S *and* T*, we define the* constraint measure[4].

$$|S \texttt{ eq } T| := 1 + |S| + |T| + \begin{cases} |T| & \text{if } \mathrm{var}(S) \wedge \mathrm{fun}(T) \\ |S| & \text{if } \mathrm{var}(T) \wedge \mathrm{fun}(S) \\ 0 & \text{otherwise} \end{cases}$$

[3] Term measures are also called *norms* in the literature on termination of constraint logic programs.

[4] Constraint measures are also called *level mappings* and *ranks* in the literature on termination of constraint logic programs.

The constraint measure summation consists of three components. The first component counts each equation as 1. The number of equations decreases when the rule `reflexivity` or `decomposition` of constants is applied. The second component $|S| + |T|$ accounts for the sizes of the arguments of the equations. It decreases when rule `decomposition` is applied. The third component adds to a variable in one argument the size of the other argument. It is introduced to handle the rule `confrontation`, where a variable is replaced by the term in its other argument. This reasoning will be made more formal in the next subsection when we compute the derivation length.

Definition 6. *For a conjunction* $\bigwedge_{i=1}^{n} S_i$ `eq` T_i *of equations (which we call problem* C*), consisting of terms* S_i *and* T_i *for* $1 \leq i \leq n$ *and* $n \in \mathbf{N}$*, we define the* problem measure $|C| := \sum_{i=1}^{n} |S_i \texttt{ eq } T_i|$*. The problem measure is extended to any conjunction of constraints by ignoring any occurrence of the built-in constraints true and false, i.e.,* $|false| := |true| := 0$*.*

Clearly, the *problem measure* is invariant to *reordering* and to *orientation* of equations. The problem measure decreases if one of its contributing constraint measures decreases. Thus, local replacements of equations, caused by a rule applications, can be treated independently.

4.2 Measure Order

Now we replace the generic order of the RT solver by a *measure order* which is defined via the term measure.

Definition 7. *The* measure order $\prec_m$ *induced by the term measure is defined by the three cases:*

1. *For two variables* X *and* Y*:* $X \prec_m Y$ *iff* $|Y| < |X|$
2. *For any variable* X *and any function term* T*:* $X \prec_m T$
3. *For two function terms* S *and* T*:* $S \prec_m T$ *iff* $|S| < |T|$

We emphasise that variables X_i and X_j are ordered by *decreasing term measure*, $X_i \prec_m X_j$ iff $|X_j| < |X_i|$ iff $j < i$, while function terms S and T are ordered by *increasing term measure*, $S \prec_m T$ iff $|S| < |T|$. The reverse ordering of variables will come handy when reasoning about the rule `confrontation`.

In the sequel, we assume the RT solver of Fig. 1 uses our measure order $\prec_m$ (c.f. Definition 7). Note that the measure order $\prec_m$ is *not* an instance of the generic term order $\prec$. For example, $f(f(f(a))) \prec_s f(a,a,a,a)$ in term-size order while $f(a,a,a,a) \prec_m f(f(f(a)))$ in measure order.

4.3 Number of Rule Applications

We will need the following inequality between the constraint measure and its arguments' term measures.

Lemma 1 *For any two terms S and T, the constraint measure is bounded by twice the sum of its term measures plus one:* $|S \texttt{ eq } T| < 2\big(1 + |S| + |T|\big)$.

Proof. Directly by Definition 5. □

The problem measure gives an upper bound on the derivation length.

Lemma 2 *Each application of one of the rules* `reflexivity`*,* `decomposition`*, and* `confrontation` *decreases the problem measure.*

Proof. We consider each rule in turn.

Application of `reflexivity`: Consider any variable X.

$$|X \texttt{ eq } X| = 1 + |X| + |X| \;>\; 0 = |\mathit{true}|$$

Application of `decomposition`: First, consider the case of two function terms with same function symbols and same arities.

$$\begin{aligned}
|f(S_1,\ldots,S_n) \texttt{ eq } f(T_1,\ldots,T_n)| &= 1 + |f(S_1,\ldots,S_n)| + |f(T_1,\ldots,T_n)| \\
&= 1 + \big(n + 2\sum_{i=1}^{n} |S_i|\big) + \big(n + 2\sum_{i=1}^{n} |T_i|\big) \\
&= 1 + \sum_{i=1}^{n} \underbrace{2(1 + |S_i| + |T_i|)}_{\underset{\text{Lemma 1}}{>} |S_i \texttt{ eq } T_i|} \\
&> \sum_{i=1}^{n} |S_i \texttt{ eq } T_i| = \left| \bigwedge_{i=1}^{n} S_i \texttt{ eq } T_i \right|
\end{aligned}$$

Note that decomposition of constants (i.e., null-ary function symbols) is also covered. Second, when the two function terms have different function symbols or different arities ($f \neq g$ or $m \neq n$) there is a *clash*. Then the RT solver immediately returns *false*.

$$\begin{aligned}
|f(S_1,\ldots,S_m) \texttt{ eq } g(T_1,\ldots,T_n)| &= 1 + |f(S_1,\ldots,S_m)| + |f(T_1,\ldots,T_n)| \\
&> 0 = |\mathit{false}|
\end{aligned}$$

Application of `confrontation`: We consider all three cases in turn (due to the restrictions by the guard there is no fourth case). For a variable X and two arbitrary terms T_1 and T_2, the guard requires $X \prec_m T_1$ and $T_1 \preceq_m T_2$.

- For two variables T_1 and T_2, we have $|X| > |T_1|$ (because $X \prec_m T_1$)

$$|\texttt{X eq } T_2| = 1 + |X| + |T_2| > 1 + |T_1| + |T_2| = |T_1 \texttt{ eq } T_2|$$

- For a variable T_1 and a function term T_2, we have $|X| > |T_1|$ (because $X \prec_m T_1$)

$$|\texttt{X eq } T_2| = 1 + |X| + 2|T_2| > 1 + |T_1| + 2|T_2| = |T_1 \texttt{ eq } T_2|$$

– Finally, for two function terms T_1 and T_2, we have $|T_1| \leq |T_2|$ (because $T_1 \preceq_m T_2$)

$$|\texttt{X eq } T_2| = 1+|X|+2|T_2| > 1+|T_2|+|T_2| \geq 1+|T_1|+|T_2| = |T_1 \texttt{ eq } T_2|$$

In all three cases, application of the corresponding rule decreases the problem measure. □

Lemma 3 *The number of rule applications (for all four rules) is bounded by twice the problem measure.*

Proof. By Lemma 2, the problem measure is an upper bound for the number of rule applications of `reflexivity`, `decomposition`, or `confrontation`. The problem measure is invariant to orientation of equations. As `orientation` can apply at most *once* to each available constraint, there are at most *twice* as many rule applications by all four rules than the problem measure. □

4.4 Tightness of the Problem Measure

To exhibit the worst-case, the query should decrease the problem measure as little as possible, i.e. the strict inequalities in the proof of Lemma 2 should be as tight as possible. We can see that we should use as few variables as possible; avoid a clash; make sure that after decomposition, the new equations are between a variable and a function term; confront terms with the same measure if possible.

Definition 8. *For the variable X and the binary function symbol f we define the following, mutually recursive terms for all natural numbers $n \in \mathbf{N}$.*

$$\mathcal{U}_n := \begin{cases} X & \text{if } n = 0 \\ f(\mathcal{L}_{n-1}, X) & \text{otherwise} \end{cases} \qquad \mathcal{L}_n := \begin{cases} X & \text{if } n = 0 \\ f(X, \mathcal{U}_{n-1}) & \text{otherwise} \end{cases}$$

For example, for $n = 4$ we have $\mathcal{U}_4 = f(f(X, f(f(X, X), X)), X)$ and $\mathcal{L}_4 = f(X, f(f(X, f(X, X)), X))$.

Lemma 4 (Properties of $\mathcal{U}_n$ and $\mathcal{L}_n$) *For $n \in \mathbf{N}$:*

1. $\#\mathcal{U}_n = \#\mathcal{L}_n$ *and* $|\mathcal{U}_n| = |\mathcal{L}_n|$.
2. *The term size is* linear*:* $\#\mathcal{U}_n = 2n + 1$.
3. *The term measure is* exponential*:* $|\mathcal{U}_n| \geq 2^n$.
4. $\mathcal{L}_n \preceq_m \mathcal{U}_n$ *and* $\mathcal{U}_n \preceq_m \mathcal{L}_n$

Proof. The easy inductions are omitted for lack of space.

Because the terms $\mathcal{L}_n$ and $\mathcal{U}_n$ are equivalent w.r.t. to measure order $\prec_m$ we can give a computation, that produces $\mathcal{L}_{n-1}$ `eq` $\mathcal{U}_{n-1}$ from $\mathcal{L}_n$ `eq` $\mathcal{U}_n$. In detail, we provide a query consisting of such equations which has exponential derivation length.

Lemma 5 (Exponential query) *For $n \in \mathbf{N}^+$ consider the following query $Q(n)$.*

$$\left(\bigwedge_{i=1}^{n} X \ \texttt{eq}\ \mathcal{L}_i\right) \wedge X \ \texttt{eq}\ \mathcal{U}_n \wedge X \ \texttt{eq}\ \mathcal{L}_n$$

Query $Q(n)$ has quadratic size $\#Q(n) = O(n^2)$. There exists a computation in standard semantics for $Q(n)$ which produces exponentially many equations: precisely, 2^{n+1} equations X eq X are produced.

We delay the proof of Lemma 5 and introduce a *sub-computation* $S(n)$ (for a given $n \in \mathbf{N}^+$). Sub-computation $S(n)$ can be applied to states that contain c copies[5] of the conjunction $X \ \texttt{eq}\ \mathcal{U}_n \wedge X \ \texttt{eq}\ \mathcal{L}_n$ plus one additional *catalyst* copy of $X \ \texttt{eq}\ \mathcal{L}_n$ for some $c \in \mathbf{N}^+$. Note that using standard semantics, we are free to select the order in which rules are applied. $S(n)$ consists of three phases where rule applications double the number of non-catalyst constraints.

Phase 1
Application (for c times) of `confrontation` between $X \ \texttt{eq}\ \mathcal{U}_n$ and $X \ \texttt{eq}\ \mathcal{L}_n$:

$$X \ \texttt{eq}\ \mathcal{U}_n \wedge X \ \texttt{eq}\ \mathcal{L}_n \mapsto_{\texttt{confront.}} X \ \texttt{eq}\ \mathcal{U}_n \wedge \mathcal{U}_n \ \texttt{eq}\ \mathcal{L}_n$$

Note that a copy of the constraint $X \ \texttt{eq}\ \mathcal{L}_n$ remains unchanged. The number of constraints is unchanged in Phase 1.

Phase 2
Application (for c times) of `confrontation` between $X \ \texttt{eq}\ \mathcal{L}_n$ and $X \ \texttt{eq}\ \mathcal{U}_n$:

$$X \ \texttt{eq}\ \mathcal{L}_n \wedge X \ \texttt{eq}\ \mathcal{U}_n \mapsto_{\texttt{confront.}} X \ \texttt{eq}\ \mathcal{L}_n \wedge \mathcal{L}_n \ \texttt{eq}\ \mathcal{U}_n$$

The number of constraints is unchanged in Phase 2.

Phase 3
Application of `decomposition` and `orientation` to c copies of $\mathcal{L}_n \ \texttt{eq}\ \mathcal{U}_n$ and to c copies of $\mathcal{U}_n \ \texttt{eq}\ \mathcal{L}_n$:

$$\mathcal{L}_n \ \texttt{eq}\ \mathcal{U}_n \mapsto_{\texttt{decomp.}} \mapsto^* X \ \texttt{eq}\ \mathcal{U}_{n-1} \wedge X \ \texttt{eq}\ \mathcal{L}_{n-1}$$
$$\mathcal{U}_n \ \texttt{eq}\ \mathcal{L}_n \mapsto_{\texttt{decomp.}} \mapsto^* X \ \texttt{eq}\ \mathcal{U}_{n-1} \wedge X \ \texttt{eq}\ \mathcal{L}_{n-1}$$

Each of the application removes *one* equation while producing *two* new equations. The number of non-catalyst constraints doubles in Phase 3.

Example 3. The computation steps of sub-computation $S(2)$ applied on the query $\left(\bigwedge_{i=1}^{2} X \ \texttt{eq}\ \mathcal{L}_i\right) \wedge X \ \texttt{eq}\ \mathcal{U}_2 \wedge X \ \texttt{eq}\ \mathcal{L}_2$ are given given in Fig. 2. The *catalyst part* $\bigwedge_{i=1}^{2} X \ \texttt{eq}\ \mathcal{L}_i$ (the first two constraints in each state) is unchanged. If we run sub-computation $S(1)$ on the answer given by $S(2)$, we first apply rule `confrontation` for four times on copies of $X \ \texttt{eq}\ f(X,X) \wedge X \ \texttt{eq}\ f(X,X)$. Finally, the generated four copies of $f(X,X) \ \texttt{eq}\ f(X,X)$ are simplified to eight copies of $X \ \texttt{eq}\ X$ by repeated application of `decomposition`.

[5] Remember that CHR conjunctions are considered as *multi-sets* of atomic constraints.

```
        X eq f(X,X), X eq f(X,f(X,X)), X eq f(f(X,X),X), X eq f(X,f(X,X))
↦co.    X eq f(X,X), X eq f(X,f(X,X)), X eq f(f(X,X),X), f(f(X,X),X) eq f(X,f(X,X))
↦co.    X eq f(X,X), X eq f(X,f(X,X)), f(X,f(X,X)) eq f(f(X,X),X), f(f(X,X),X) eq f(X,f(X,X))
↦de.↦*  X eq f(X,X), X eq f(X,f(X,X)), X eq f(X,X), X eq f(X,X), f(f(X,X),X) eq f(X,f(X,X))
↦de.↦*  X eq f(X,X), X eq f(X,f(X,X)), X eq f(X,X), X eq f(X,X), X eq f(X,X), X eq f(X,X)
```

Fig. 2. Sub-computation $S(2)$ applied on $\left(\bigwedge_{i=1}^{2} X \texttt{ eq } \mathcal{L}_i\right) \wedge X \texttt{ eq } \mathcal{U}_2 \wedge X \texttt{ eq } \mathcal{L}_2$

Lemma 6 *Application of $S(n)$ replaces* each copy *of the conjunction X `eq` $\mathcal{U}_n \wedge X$ `eq` $\mathcal{L}_n$ by* two copies *of the conjunction X `eq` $\mathcal{U}_{n-1} \wedge X$ `eq` $\mathcal{L}_{n-1}$ for $n \in \mathbf{N}^+$. The catalyst constraint X `eq` $\mathcal{L}_n$ remains unchanged while the number of rewritten non-catalyst equations* doubles.

Proof. We apply sub-computation $S(n)$ using the *catalyst* copy of X `eq` $\mathcal{L}_n$ on the c copies of the conjunction X `eq` $\mathcal{U}_n \wedge X$ `eq` $\mathcal{L}_n$. Applying rules according to the phases of the $S(n)$ we create $2c$ copies of the conjunction X `eq` $\mathcal{U}_{n-1} \wedge X$ `eq` $\mathcal{L}_{n-1}$. □

Proof (Lemma 5). We apply $S(i)$ repeatedly starting with the initial query $(\bigwedge_{i=1}^{n} X \texttt{ eq } \mathcal{L}_i) \wedge X \texttt{ eq } \mathcal{U}_n \wedge X \texttt{ eq } \mathcal{L}_n$. Formally, we use induction on the sequential application of $S(n), S(n-1), \ldots, S(1)$ which is possible because the *catalyst part* $\bigwedge_{i=1}^{n} X \texttt{ eq } \mathcal{L}_i$ remains unchanged. Doubling the number of copies of the rewritten (non-catalyst) constraints each time we apply $S(i)$, we arrive at 2^{n+1} (non-catalyst) constraints `X eq X`. □

The sub-computation $S(n)$ of the exponential query crucially relies on a scheduling of the rules such that all the phases are possible. We can simulate this instance of standard operational semantics by a CHR program for refined semantics (sources available [1]). When we improve the complexity of the solver to quadratic in Section 5, we will see that such a scheduling is not possible for flat constraints. Actually it seems that it is impossible under the refined semantics, thus impossible in any practical sequential implementation of CHR that currently exists. Thus the worst-case complexity of the RT solver (instantiated with a corrected order) for refined semantics is still an open problem.

4.5 Worst-Case Time and Space Complexity

Combining our results from the preceeding two subsections, we can now give our main result: The RT solver with measure order $\prec_m$ has exponential space and (hence) exponential time complexity under the standard semantics of CHR.

Lemma 7 *Any conjunction of equations C with size $\#C$ has problem measure $|C| = O(2^{\#C})$.*

Proof. Skipped for lack of space. □

As the problem measure is (at most) exponential in the size of the problem, the derivation length is (at most) exponential (by Lemma 3). We constructed a

query of problem size $O(n^2)$ which produces more than 2^n equations for a specific computation strategy under the standard semantics. Therefore, our bound for the derivation length and the resulting exponential worst-case complexity are tight.

Theorem 1 (Exponential Complexity) *For the RT solver of Fig. 1 with measure order $\prec_m$, the number of rule applications is exponential in the size of the problem in the worst-case.*

Proof. By Lemma 3 and Lemma 7, the number of rule applications is at most exponential in the problem size. By Lemma 5 this upper bound is tight. □

As $\mathcal{L}_n \preceq_s \mathcal{U}_n$ and $\mathcal{U}_n \preceq_s \mathcal{L}_n$ in term-size order, $S(i)$ is also applicable and the solver with term-size order has *at least* exponential worst-case complexity under the standard semantics.

5 Quadratic Complexity

We can improve the worst-case time and space complexity of the CHR rational tree solver from exponential to quadratic by simply requiring that equations are in flat normal form when the problem is given. A term can be flattened by performing the opposite of variable elimination. Each sub-term is replaced by a new variable that is equated with the replaced expression.

Definition 9. *A conjunction of constraints is in* flat normal form *if each argument of each constraint contains at most one function symbol, i.e., it is either a variable or a function applied to variables.*

For flattening it suffices to traverse the constraints of the problem once and to replace nested function symbols by a new variable and a new equation with that variable. (A function symbol is *nested* if it occurs inside another term.) For our proofs it is not necessary that the flattening function produces the minimal number of equations.

Definition 10. *The* flattening function $[.]$ *transforms the syntactic equality constraints into an equivalent conjunction of flattened equations. For a conjunction of constraints $\bigwedge_{i=1}^{n} S_i \ \texttt{eq}\ T_i$, we introduce new variables $X_1, \ldots, X_n$ and define*

$$\left[\bigwedge_{i=1}^{n} S_i \ \texttt{eq}\ T_i\right] := \bigwedge_{i=1}^{n} \left([X_i \ \texttt{eq}\ S_i]_1 \wedge [X_i \ \texttt{eq}\ T_i]_1\right)$$

For an atomic constraint $X \ \texttt{eq}\ T$, we define the auxiliary function $[.]_1$ as follows (with new variables $X_1, \ldots, X_n$)

$$[X \ \texttt{eq}\ T]_1 := \begin{cases} X \ \texttt{eq}\ T & \text{if } \mathrm{var}(T) \\ X \ \texttt{eq}\ f(X_1, \ldots, X_n) \wedge \left(\bigwedge_{i=1}^{n} [X_i \ \texttt{eq}\ T_i]_1\right) & \text{if } T = f(T_1, \ldots, T_n) \end{cases}$$

Lemma 8 *The size of the flattened problem $\#[C]$ is linear in the problem size, i.e., $\#[C] = O(\#C)$. Also the number of new variables and the number of new equations is linear in the problem size.*

Proof. ¿From Definition 10 we can see that the variables and function symbols of the original problem are kept. In addition, new variables are introduced for each original equation and for each nested function symbol and variable. Each new variable occurs twice. The number of original equations and of nested function symbols and variables is bounded by the number of function symbols and variables in the problem, i.e. by the problem size, because the arguments of an equation are not-nested symbols, either variables or outermost function symbols.

Therefore the size of the flattened problem is at most three times the size of the original problem size. Also, the number of new variables is bounded by the problem size, and thus the number of variables is linear in the problem size. Since the flattened equations have a new variable as first argument and each original equation is replaced by two new ones, the number of equations is at most twice the problem size. □

Lemma 9 *The flattening of a problem C can be done in linear time and space w.r.t. the problem size $\#C$.*

Proof. By Lemma 8 and by Definition 10 of the flattening function [.]. □

In flattened problems, the problem measure is quadratic in the problem size, while for general problems, it was exponential. The improvement is due to the fact that the depth of flat terms is at most one.

Lemma 10 *Given a* flat *problem C with at least one variable, its measure $|C| = O(v\,\#C)$ is bounded by the problem size and number of variables v.*

Proof. Analogous to the problem size, the problem measure is defined as the sum of the measures of the atomic constraints' arguments in the problem. Therefore a case analysis on the structure of flat terms suffices.

1. For any variable $|X_i| = i \leq v\,\#X_i$ as $i{\leq}v$ and $\#X_i{=}1$.
2. For any flat function term $T = f(X_{j_1}, \ldots, X_{j_n})$, we have $|T| = n + 2\sum_{j=1}^{n} j_i \leq n + 2\sum_{j=1}^{n} v = (2v+1)n \leq (2v+1)\,\#T$ as $\#T{=}n{+}1$.

By Definitions 5 and 6 we conclude $|C| = O(v\,\#C)$. □

Now we can prove that the overall complexity is quadratic in the problem size when the problem is in flat normal form.

Theorem 2 (Quadratic Complexity) *For the RT solver of Fig. 1 with measure order $\prec_m$, the number of rule applications is quadratic in the problem size in the worst-case if the problem is in flat normal form.*

Proof. The number of rule applications is bounded by the problem measure by Theorem 1. Given a problem C, by Lemmas 8 and 10, the measure size of the problem in flat normal form is quadratic in the problem sizes: $\big|[C]\big| = O(v\,\#[C]) = O(\#^2[C]) = O(\#^2 C)$. □

Analogous proofs of quadratic time and space complexity can be given for terms with *bounded depth*, only the constant factors increase.

For flattened problems, the rule `decomposition` either fails due to a clash or produces equations between variables only. Flat terms that do not clash have the same term size. So it does not matter how function terms of same size are ordered by the instance of the generic term order $\prec$. Therefore Theorem 2 is also applicable to any instance of the generic term order used in the RT solver. We conjecture that even the Prolog built-in term order `@<` is sufficient, since functional terms can be ordered arbitrarily without changing the sizes of the equations involved.

6 Conclusion

The complexity of the classic CHR rational tree equation solver [7, 4, 8, 14] was an open problem. For termination, the solver relies on a generic order between terms that must fulfil some properties. The standard implementation of the solver that is included in many CHR libraries uses the built-in Prolog term order that does not respect all properties. We gave an example for non-termination of that solver.

Our main result shows that there exists a term order for the classic CHR rational tree equation solver that leads to *exponential worst-case time and space complexity* in the size of the problem under the standard CHR semantics (that does not constrain the order of rule applications). This complexity bound is *tight*. This term order, however, is not an instance of the generic term order. It is based on a term measure that is exponential in the depth of the term.

Since the generic term order usually required for termination of the RT solver and the measure order we have defined in this paper are incompatible, we conjecture that there is a more general generic order that subsumes these orders, and that this order is based on the sub-term relation. We are also interested in other measures based on term-depth or explicit exponentiation like $2^{\#T}$.

Our complexity proof does not apply to actual implementations of the RT solver that usually rely on the refined CHR semantics, but it implies that under standard semantics, their complexity is *at least* exponential when the term-size instance of the generic term order is used. It is still an open question, whether the complexity of the solver with generic term order using other order instances and/or using the refined semantics is polynomial or not. However, this question is not so burning anymore in the light of our following result.

We improved the complexity of the solver to be *quadratic for any term order* (including the built-in Prolog one) under *standard and refined semantics* by simply requiring that equations are in flat normal form before solving them. Since any conjunction of equations can be flattened in linear time and space, this gives an efficient polynomial algorithm.

Since there is no performance penalty in time and space complexity (except constant factors) when using CHR [16], one may be interested in a *quasi-linear* solver. Such a solver is implementable by a straightforward combination [2] of the RT solver with the union-find algorithm in CHR [15, 6] that will handle all

equations between variables. During this line of work, the CHR RT solver should be more thoroughly compared to existing classical unification algorithms (c.f. Section 1). This includes to check if our proof methods apply to implementations of these algorithms as well.

CHR program sources for this paper are available [1].

References

1. CHR program sources for this paper (SICStus Prolog 3.11), 2006. `http://www.informatik.uni-ulm.de/pm/index.php?id=133`.
2. S. Abdennadher and T. Frühwirth. Integration and optimization of rule-based constraint solvers. In M. Bruynooghe, editor, *LOPSTR 2003*, volume 3018 of *LNCS*, pages 198–213. Springer, 2003.
3. A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, London, 1982.
4. T. Frühwirth. Theory and practice of Constraint Handling Rules, Special issue on constraint logic programming. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
5. T. Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In D. Fensel, F. Giunchiglia, D. L. McGuinness, and M.-A. Williams, editors, *8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 547–557. Morgan Kaufmann, 2002.
6. T. Frühwirth. Parallelizing union-find in Constraint Handling Rules using confluence. In M. Gabbrielli and G. Gupta, editors, *ICLP*, volume 3668 of *LNCS*, pages 113–127. Springer, 2005.
7. T. Frühwirth and S. Abdennadher. *Constraint-Programmierung*. Springer, 1997.
8. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.
9. J. Herbrand. *Recherches sur la théorie de la demonstration*. PhD thesis, Université de Paris, France, 1930.
10. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980.
11. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.
12. M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.
13. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
14. T. Schrijvers and T. Frühwirth. Constraint Handling Rules (CHR) web page, 2006. `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`.
15. T. Schrijvers and T. Frühwirth. Optimal union-find in Constraint Handling Rules, programming pearl. *Theory and Practice of Logic Programming (TPLP)*, 6(1&2):213–224, 2006.
16. J. Sneyers, T. Schrijvers, and B. Demoen. The computational power and complexity of Constraint Handling Rules. In T. Schrijvers and T. Frühwirth, editors, *CHR 2005*, pages 3–18, Sitges, Spain, 2005. Report CW421, K.U. Leuven, Belgium.
17. R. E. Tarjan and J. Van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.

A compositional Semantics for CHR with propagation rules

Maurizio Gabbrielli[1], Maria Chiara Meo[2], and Paolo Tacchella[1]

[1] Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura A. Zamboni 7, 40127 Bologna, Italy
gabbri@cs.unibo.it, Paolo.Tacchella@cs.unibo.it
[2] Dipartimento di Scienze, Università di Chieti
Viale Pindaro 42, 65127 Pescara, Italy
cmeo@unich.it

Abstract.
Constraint Handling Rules (CHR) are a committed-choice declarative language which has been designed for writing constraint solvers.
In [8] a trace based compositional semantics for CHR has been defined. The reference operational semantics for such a compositional model is the original, naive, operational semantics for CHR which, due to the propagation rule, admits trivial non-termination.
In this paper we extend the work of [8] by considering a more refined operational semantics which avoids trivial non-termination.

1 Introduction

Constraint Handling Rules (CHR) [10, 11] are a committed-choice declarative language which has been specifically designed for writing constraint solvers. CHR permit to easily introduce specific user-defined constraints and the related solver into a host language. They have received a considerable attention, both from the practical and the theoretical side and various languages support CHR implementation: Prolog [13, 15], HAL [12], Haskell and Java [2]. A *CHR program* is a set of *simplification*, *simpagation* and *propagation* rules. A naive use of the last rule can introduce unnecessary infinite computation as the original CHR semantics [11] does. More refined semantics [1, 9] avoid this problem considering the history of computation.

All these semantics, as well some other versions defined elsewhere, are not compositional w.r.t. the conjunction of atoms in a goal. This is somehow surprising, considering CHR both from the logic programming and the concurrency theory perspective. In the first case, in fact, the reference semantics (least Herbrand model) as well as other more refined semantics (e.g. the s-semantics) enjoy this property. When considering CHR as a (committed choice) concurrent language the situation is even more surprising, as the conjunction of atoms can naturally be considered as parallel composition and most semantics are indeed compositional w.r.t. parallel composition (as well as w.r.t. all the other operators of the language). Indeed compositionality of a semantics is in general a very desirable feature, as it permits to manage partially defined components and it

can be the basis to define incremental and modular tools for software analysis and verification. For these reasons in [8] a fix-point, and-compositional semantics for CHR was defined, which allows to retrieve the semantics of a conjunctive query from the semantics of its components. This compositional semantics uses semantic structures based on traces, similarly to what has been done for data-flow languages [14], imperative concurrent languages [7] and concurrent constraint languages [5]. However, as discussed in [8], due to the presence of multiple heads in the case of CHR one needs traces more complicated than those used in the above mentioned works.

The semantics defined in [8] uses as reference operational semantics that one defined in [11]. In this paper we extend the work of [8] by considering as reference semantics one [9] which avoids trivial infinite computations using a token store, that controls the number of applications of a propagation rule. This further complicates the structure of traces and affects the composition of sequences. We must ensure that a propagation rule is not applied twice to the same constraints present in both of sequences. The correctness of this new semantics is proved w.r.t. a slightly different notion of observables than the one in [8] since now token store is also considered.

The remaining of this paper is organized as follows. Next section introduces some preliminaries about CHR. Section 3 contains the definition of the compositional semantics, while section 4 presents the compositionality and correctness results. Section 5 concludes by discussing directions for future works.

2 Preliminaries

In this section we introduce CHR syntax using an example, its theoretical semantics ω_t, the notations used and some definitions. There are two kinds of constraints: the built-in ones, defined by $c ::= a|c \wedge c|\exists_x a$ handled by an existing solver, for which we assume given a theory CT, which describes their meaning, with a atom; the CHR ones.

We use c, d to denote built-in constraints, g, h, k to denote CHR constraints and a, b to denote both built-in and user-defined constraints (we will call these generically constraints). The capital versions of these notations will be used to denote multisets of constraints. Furthermore we denote by $\mathcal{U}$ the set of user-defined constraints.

We will often use "," rather than $\wedge$ to denote conjunction and we will often consider a conjunction of atomic constraints as a multiset of atomic (possibly identified) constraints. The notation $\exists_{-V}\phi$, where V is a set of variables, denotes the existential closure of a formula ϕ with the exception of the variables V which remain unquantified. $Fv(\phi)$ denotes the free variables appearing in ϕ. We denote by $\cdot$ the concatenation of sequences, by ε the empty sequence and by $\setminus$ the set difference operator. Moreover, given a goal G, we denote by $\tilde{G}$ one of the possible identified versions of G. For the sake of simplicity, we will omit the superscript $\tilde{}$, when clear from the context.

We are now ready to define the CHR syntax.

Definition 1 (Syntax). *[9, 11] A CHR* simplification *rule has the form $r@H \Leftrightarrow C \mid B$; a CHR* propagation *rule has the form $r@H \Rightarrow C \mid B$; a CHR* simpagation *rule has the form $r@H_1 \setminus H_2 \Leftrightarrow C \mid B$ where r is a unique identifier of a rule, H, H_1 and H_2 are sequences of user-defined constraints, C is a multiset of built-in constraints and B is a possibly empty multi-set of (built-in and user-defined) constraints.*

Solve$_{\omega_t}$	$\dfrac{CT \models c \wedge d \leftrightarrow d' \text{ and c is a built-in constraint}}{\langle \{c\} \uplus G, S, d, T\rangle_n \longrightarrow_{\omega_t} \langle G, S, d', T\rangle_n}$
Introduce$_{\omega_t}$	$\dfrac{\text{h is a user-defined constraint}}{\langle \{h\} \uplus G, S, c, T\rangle_n \longrightarrow_{\omega_t} \langle G, \{h\#n\} \cup S, c, T\rangle_{n+1}}$
Apply$_{\omega_t}$	$\dfrac{r@H_1' \setminus H_2' \Leftrightarrow C \mid B \in P \quad x = Fv(H_1', H_2') \quad CT \models c \rightarrow \exists_x((chr(H_1, H_2) = H_1', H_2') \wedge C)}{\langle G, \{H_1\} \cup \{H_2\} \cup S, c, T\rangle_n \longrightarrow_{\omega_t} \langle B \uplus G, \{H_1\} \cup S, (chr(H_1, H_2) = (H_1', H_2')) \wedge c, T'\rangle_n}$ where $r@id(H_1, H_2) \notin T$ and $T' = T \cup \{r@id(H_1, H_2)\}$

Table 1. The standard transition system Tx for CHR

The *simpagation* rule behaves as a simplification or propagation one respectively if H_1 or H_2 is empty, with $(H_1, H_2) \neq \emptyset$. A *CHR program* is a finite set of CHR simplification, propagation and simpagation rules. A *CHR goal* is a set of (both identified user-defined and built-in) constraints. $Goals$ is the set of all goals.

Example 1. After the definition we propose an example of program: the CHR that encode less-than-or-equal constraint [11]:

$rfl \ @ \ X =< Y \Leftrightarrow X = Y \mid \texttt{true}$	reflexivity
$asx \ @ \ X =< Y, Y =< X \Leftrightarrow X = Y$	antisymmetry
$trs \ @ \ X =< Y, Y =< Z \Rightarrow X =< Z$	transitivity
$idp \ @ \ X =< Y \backslash X =< Y \Leftrightarrow \texttt{true}$	idempotence

We describe now the operational semantics of CHR introduced in [9] by using a transition system $Tx = (Conf_t, \longrightarrow_{\omega_t})$. Configurations in $Conf_t$ are tuples of the form $\langle G, S, c, T\rangle_n$ where G, the goal is a multiset of constraints to be executed. The CHR constraint store S is the set of identified CHR constraints that can be matched with rules in the program P. An identified CHR constraint $g\#i$ is a CHR constraint g, associated with some unique integer i. This number serves to differentiate among copies of same constraint. We introduce functions *chr(g#i)=g* and *id(g#i)=i*, and extend them to sets and sequences of identified CHR constraints in the obvious manner. The *propagation history* T is a set of tokens of the form $r@i_1, \ldots, i_m$, where r is the name of the applied rule and i_s, with $1 \leq s \leq m$ the sequence of unique identifiers associated to the constraints to which the head of rule is applied. This is necessary to prevent trivial nontermination for propagation rules: a propagation rule is allowed to fire on a sequence of constraints only if the same sequence of constraints have been not used to fire the rule before. Finally the counter n represents the next free integer which can be used to number a CHR constraint.

Given a goal G, the *initial configuration* is: $\langle G, \emptyset, \texttt{true}, \emptyset\rangle_1$ and consists of a goal G, an empty CHR constraint, an empty built-in constraint and an empty set of tokens.

A *final configuration* has either the form: $\langle G, S, \texttt{false}, T\rangle_n$ when it is *failed*, i.e. when it contains an inconsistent built-in constraint store represented by the unsatisfiable

constraint `false`, or it has the form $\langle \emptyset, S, c, T \rangle_n$ when it is successfully terminated since there are no applicable rules.

Given a program P, the transition relation $\longrightarrow_{\omega_t} \subseteq Conf_t \times Conf_t$ is the least relation satisfying the rules in Table 1 (for the sake of simplicity, we omit indexing the relation with the name of the program). The **Solve**$_{\omega_t}$ transition allows to update the constraint store by taking into account a built-in constraint contained in the goal. Without loss of generality, we will assume that $Fv(d') \subseteq Fv(d) \cup Fv(c)$. The **Introduce**$_{\omega_t}$ transition is used to move a user-defined constraint from the goal to the CHR constraint store, where it can be handled by applying CHR rules. The **Apply**$_{\omega_t}$ transition rewrites user-defined constraints in CHR store using the rules of program. It assumes that all variables appearing in a program clause are renamed apart with fresh ones to avoid variable names clashes.

Following definition introduces a kind of "answers" that a final state can give.

Definition 2 (Data sufficient answers). *Let P be a program and let G be a goal. The set $\mathcal{SA}_P(G)$ of data sufficient answers for the query G in the program P is defined as follows*

$$\begin{array}{rl} \mathcal{SA}_P(G) = & \{\langle \exists_{-Fv(G)} d \rangle \mid \langle G, \emptyset, \mathtt{true}, \emptyset \rangle_1 \longrightarrow^*_{\omega_t} \langle \emptyset, \emptyset, d, T \rangle_n \not\longrightarrow_{\omega_t}\}. \\ & \cup \\ & \{\langle \mathtt{false} \rangle \mid \langle G, \emptyset, \mathtt{true}, \emptyset \rangle_1 \longrightarrow^*_{\omega_t} \langle G', K, \mathtt{false}, T \rangle_n\}. \end{array}$$

3 A compositional trace semantics

Given a program P, we say that a semantics $\mathcal{S}_P$ is and-compositional if $\mathcal{S}_P(A, B) = \mathcal{C}(\mathcal{S}_P(A), \mathcal{S}_P(B))$ for a suitable composition operator $\mathcal{C}$ which does not depend on the program P. The presence of multiple heads in CHR makes the semantics, which associates to a program P the function $\mathcal{SA}_P$, not and-compositional. In fact the goals, which have the same input/output behavior, can behave differently when composed with other goals. Consider for example the program P consisting of the single rule $r@g, h \Leftrightarrow true|c$ (where c is a built-in constraint). According to Definition 2 we have that $\mathcal{SA}_P(g) = \mathcal{SA}_P(k) = \emptyset$, while $\mathcal{SA}_P(g, h) = \{\langle \exists_{-Fv(g,h)} c \rangle\} \neq \emptyset = \mathcal{SA}_P(k, h)$.

In order to solve the problem exemplified above we intend to develop a new transition system that allows one to generate the sequences appearing in the compositional model by using a standard fix-point construction. This system collects in the semantics also the "missing" parts of heads which are needed in order to proceed with the computation. For example, when considering the program P above, we should be able to state that the goal g produces the constraint c, provided that the external environment (i.e. a conjunctive goal) contains the user-defined constraint h. When composing (by using a suitable notion of composition) such a semantics with that one of a goal which contains h we can verify that the "assumption" h is satisfied and therefore obtain the correct semantics for g, h. In order to model correctly the interaction of different processes we have to use sequences, analogously to what happens with other concurrent paradigms.

We modify the notion of configuration ($Conf_t$) used before merging the goal store with CHR one: we do not need to distinguish between them, so the **Introduce** rule

Solve' $\dfrac{CT \models c \wedge d \leftrightarrow d' \text{ and } c \text{ is a built-in constraint}}{\langle \{c\} \uplus G, d, T\rangle_n \longrightarrow_P^{\emptyset} \langle G, d', T\rangle_n}$

Apply' $\dfrac{r@H_1' \setminus H_2' \Leftrightarrow C \mid B \in P \quad x = Fv(H_1', H_2') \quad G \neq \emptyset \quad CT \models c \rightarrow \exists_x((chr(H_1, H_2) = H_1', H_2') \wedge C)}{\langle G \cup G', c, T\rangle_n \longrightarrow_P^{K} \langle I_{n+k}^{n+k+m}(B) \cup \{H_1\} \cup G', (chr(H_1, H_2) = (H_1', H_2')) \wedge c, T'\rangle_{n+k+m}}$

where k and m are the number of CHR constraints in K and in B respectively,
$\{G\} \cup \{I_n^{n+k}(K)\} = \{H_1\} \cup \{H_2\}$, $r@id(H_1, H_2) \notin T$ and
if $H_2 = \emptyset$ then $T' = T \cup \{r@id(H_1)\}$ else $T' = T$

Table 2. The transition system T for the compositional semantics

is now useless. On the other hand, we need the information on the new assumptions, which is added as a label of the transitions.

Thus we define a transition system $T = (Conf, \longrightarrow_P)$ where configurations in *Conf* are triples of the form $\langle \tilde{G}, c, T\rangle_n$: $\tilde{G}$ is a set of built-in and identified CHR constraints (the goal), c is a (conjunction of) built-in constraint(s) (the store), T is a set of tokens and n is an integer greater or equal to the biggest identifier used either to number a CHR constraint in $\tilde{G}$ or in a token in T.

In the following, given a goal G, with m CHR-constraints, we can define a function $I_n^{n+m}(G)$, which identifies each CHR constraints in G by associating to it a unique integer in $[n+1, m+n]$ by following a lexicographic order. The identifier association is applied both to the initial goal store, at the beginning of the derivation, than to the body of a rule during the computation steps. If $m = 0$ then $I_n^n(G) = G$.

Given a program P, the transition relation $\longrightarrow_P \subseteq Conf \times Conf \times \wp(\mathcal{U})$ is the least relation satisfying the rules in Table 2, where $\wp(A)$ denotes the set consisting of all the subsets of A: the powerset. Note that we consider only the **Solve** and **Apply** rules, as the other rule, as previously mentioned, is redundant in this context. **Solve'** is essentially the same rule as the one defined in Table 1, while the **Apply'** rule is modified to consider assumptions: when reducing a goal G by using a rule having head H, the set of assumptions $K = H \setminus G$ (with $H \neq K$) is used to label the transition. Note that since we apply the function I_n^{n+k} to the assumptions K, to each atom in K, which is not consumed by the application of the **Apply'** rule, is associate an identifier in $[n+1, n+k]$ in H_1.

As before, we assume that program rules, to be used in the new Apply' rule, use fresh variables to avoid variables name captures.

Given a goal G with m CHR-constraints, $\langle I_0^m(G), \texttt{true}, \emptyset\rangle_m$ is its *initial configuration* that consists of an identified goal $I_0^m(G)$, and two empty sets: built-in and token ones. A *final configuration* has either the form $\langle \tilde{G}, \texttt{false}, T\rangle_n$, when it is *failed*, or has the form $\langle \tilde{G}, c, T\rangle_n$ when it is successfully terminated since there are no applicable rules.

Now we propose an example about the use of the transition system for compositional semantics.

Example 2. Derivation of the goal $(C = 7, A =< B, C =< A, B =< C, B =< C)$ using the transition system of Table 2 and the program of Example 1:

$\langle\{C = 7, A =< B\#1, C =< A\#2, B =< C\#3, B =< C\#4\}, \mathtt{true}, \emptyset\rangle_4 \rightarrow^{\emptyset}$ $Solve$
$\langle\{A =< B\#1, C =< A\#2, B =< C\#3, B =< C\#4\}, C = 7, \emptyset\rangle_4 \rightarrow^{\emptyset}$ $trs@1, 3$
$\langle\{A =< C\#5, A =< B\#1, C =< A\#2, B =< C\#3, B =< C\#4\}, C = 7, \{trs@1, 3\}\rangle_5 \rightarrow^{\emptyset}$ $asx@5, 2$
$\langle\{A = C, A =< B\#1, B =< C\#3, B =< C\#4\}, C = 7, \{trs@1, 3\}\rangle_5 \rightarrow^{\emptyset}$ $Solve$
$\langle\{A =< B\#1, B =< C\#3, B =< C\#4\}, (A = C \wedge C = 7), \{trs@1, 3\}\rangle_5 \rightarrow^{\emptyset}$ $asx@1, 3$
$\langle\{B = C, B =< C\#4\}, (A = C \wedge C = 7), \{trs@1, 3\}\rangle_5 \rightarrow^{\emptyset}$ $Solve$
$\langle\{B =< C\#4\}, (B = C \wedge A = C \wedge C = 7), \{trs@1, 3\}\rangle_5$

The semantic domain of our compositional semantics is based on sequences which represent derivations obtained by the transition system in Table 2. We first consider "concrete" sequences consisting of tuples $\langle\, \tilde{G}, c, T, m, I_m^{m+k}(K), \tilde{G}', d, T', m'\rangle$, where k is the number of CHR atoms in K. Such a tuple represents exactly a derivation step $\langle\tilde{G}, c, T\rangle_m \longrightarrow_P^K \langle\tilde{G}', d, T'\rangle_{m'}$. The sequences we consider are terminated by tuples of the form $\langle\tilde{G}, c, T, n, \emptyset, \tilde{G}, c, T, n\rangle$, which represent a terminating step (see the precise definition below). Since a sequence represents a derivation, we assume that if

$$\ldots \langle\tilde{G}_i, c_i, T_i, m_i, \tilde{K}_i, \tilde{G}'_i, d_i, T'_i, m'_i\rangle \\ \langle\tilde{G}_{i+1}, c_{i+1}, T_{i+1}, m_{i+1}, \tilde{K}_{i+1}, \tilde{G}'_{i+1}, d_{i+1}, T'_{i+1}, m'_{i+1}\rangle \ldots$$

appears in a sequence, then $\tilde{G}'_i = \tilde{G}_{i+1}$, $T'_i = T_{i+1}$ and $m'_i \leq m_{i+1}$ hold.

On the other hand, the input store c_{i+1} can be different from the output store d_i produced at previous step, since we need to perform all the possible assumptions on the constraint c_{i+1} produced by the external environment in order to obtain a compositional semantics, so we can assume that $CT \models c_{i+1} \rightarrow d_i$ holds: this means that the assumption made on the external environment cannot be weaker than the constraint store produced at the previous step. This reflects the monotonic nature of computations, where information can be added to the constraint store and cannot be deleted from it. Finally note that assumptions on user-defined constraints (label K) are made only for the atoms which are needed to "complete" the current goal in order to apply a clause. In other words, no assumption can be made in order to apply clauses whose heads do not share any predicate with the current goal.

Example 3. The concrete sequence representation of derivation of Example 2:

$\langle\{C = 7, A =< B\#1, C =< A\#2, B =< C\#3, B =< C\#4\}, \mathtt{true}, \emptyset, 4, \emptyset$
$\quad \{A =< B\#1, C =< A\#2, B =< C\#3, B =< C\#4\}, C = 7, \emptyset, 4\rangle$
$\langle\{A =< B\#1, C =< A\#2, B =< C\#3, B =< C\#4\}, C = 7, \emptyset, 4, \emptyset$
$\quad \{A =< C\#5, A =< B\#1, C =< A\#2, B =< C\#3, B =< C\#4\}, C = 7, \{trs@1, 3\}, 5\rangle$
$\langle\{A =< C\#5, A =< B\#1, C =< A\#2, B =< C\#3, B =< C\#4\}, C = 7, \{trs@1, 3\}, 5, \emptyset$
$\quad \{A = C, A =< B\#1, B =< C\#3, B =< C\#4\}, C = 7, \{trs@1, 3\}, 5\rangle$
$\langle\{A = C, A =< B\#1, B =< C\#3, B =< C\#4\}, C = 7, \{trs@1, 3\}, 5, \emptyset$
$\quad \{A =< B\#1, B =< C\#3, B =< C\#4\}, (A = C, C = 7), \{trs@1, 3\}, 5\rangle$
$\langle\{A =< B\#1, B =< C\#3, B =< C\#4\}, (A = C, C = 7), \{trs@1, 3\}, 5, \emptyset$
$\quad \{B = C, B =< C\#4\}, (A = C, C = 7), \{trs@1, 3\}, 5\rangle$
$\langle\{B = C, B =< C\#4\}, (A = C, C = 7), \{trs@1, 3\}, 5, \emptyset$
$\quad \{B =< C\#4\}, (B = C, A = C, C = 7), \{trs@1, 3\}, 5\rangle$
$\langle\{B =< C\#4\}, (B = C, A = C, C = 7), \{trs@1, 3\}, 5, \emptyset$
$\quad \{B =< C\#4\}, (B = C, A = C, C = 7), \{trs@1, 3\}, 5\rangle$

The set of the above described "concrete" sequences, which represent derivation steps performed by using the new transition system, is denoted by $\mathcal{S}eq$.

From these concrete sequences we extract some more abstract sequences which are the objects of our semantic domain. If $\langle \tilde{G}, c, T, m, \tilde{K}, \tilde{G}', d, T', m' \rangle$ is in a sequence $\delta \in \mathcal{S}eq$ (except the last one) we extract a tuple of the form $\langle c, \tilde{K}, \tilde{H}, d \rangle$ with c and d input and output store respectively, the assumptions $\tilde{K}$ and the stable atoms $\tilde{H}$: the restriction of goal $\tilde{G}$ to the identified constraints, that will not be used more in δ to fire a rule. Output goal $\tilde{G}'$ is not considered any more. Intuitively $\tilde{H}$ contains those atoms which are available for satisfying assumptions of other goals, when composing two different sequences (representing two derivations of different goals). If $\langle c_i, \tilde{K}_i, \tilde{H}_i, d_i \rangle \langle c_{i+1}, \tilde{K}_{i+1}, \tilde{H}_{i+1}, d_{i+1} \rangle$ is in a sequence we also assume that $\tilde{H}_i \subseteq \tilde{H}_{i+1}$ holds, since these atoms which will not be rewritten in the derivation can only augment. Moreover, if $\langle \tilde{G}, c, T, m, \emptyset, \tilde{G}, c, T, m \rangle$ is the last tuple in δ, we extract a tuple of the form $\langle c, \tilde{G}, T \rangle$ where we consider as before the input store c (the output store is equal), the input goal $\tilde{G}$ and token store.

We then define formally the semantic domain as follows.

Definition 3 (Sequences). *The semantic domain $\mathcal{D}$ containing all the possible sequences is defined as the set*

$$\begin{aligned}\mathcal{D} = \{ & \langle c_1, K_1, H_1, d_1 \rangle \langle c_2, K_2, H_2, d_2 \rangle \dots \langle c_m, H_m, T_m \rangle \mid \\ & m \geq 1, \text{ for each } j,\ 1 \leq j \leq m \text{ and for each } i, 1 \leq i \leq m-1, \\ & H_j \text{ and } K_i \text{ are sets of identified CHR constraints,} \\ & T_m \text{ is a set of token, } H_i \subseteq H_{i+1} \text{ and } c_i, d_i \text{ are built-in constraints} \\ & \text{such that } CT \models d_i \rightarrow c_i \text{ and } CT \models c_{i+1} \rightarrow d_i \text{ hold} \}.\end{aligned}$$

In order to define our semantics we need two more notions: an abstraction operator α, which extracts from the concrete sequences in $\mathcal{S}eq$ (representing exactly derivation steps) the sequences used in our semantic domain, and the concept of stable atoms.

Definition 4 (Abstraction and Stable atoms). *Let*

$$\delta = \langle G_1, c_1, T_1, n_1, K_1, G_2, d_1, T_2, n_1' \rangle \dots \langle G_m, c_m, T_m, n_m, \emptyset, G_m, c_m, T_m, n_m \rangle$$

be a sequence of derivation steps where we assume that the CHR atoms are identified. We say that an identified atom $g\#l$ is stable in δ if $g\#l$ appears in G_1 and the identifier l does not appear in $T_j \setminus T_1$, for each $1 \leq j \leq m$. The abstraction operator $\alpha : \mathcal{S}eq \rightarrow \mathcal{D}$ is then defined inductively as

$$\begin{aligned}&\alpha(\langle G, c, T, n, \emptyset, G, c, T, n \rangle) = \langle c, G, T \rangle \\ &\alpha(\langle G_1, c_1, T_1, n_1, K_1, G_2, d_1, T_2, n_1' \rangle \cdot \delta') = \langle c_1, K_1, H, d_1 \rangle \cdot \alpha(\delta').\end{aligned}$$

where H is the set consisting of all the identified atoms A such that A is stable in $\langle G_1, c_1, T_1, n_1, K_1, G_2, d_1, T_2, n_1' \rangle \cdot \delta'$.

We present an example about α function that extracts from concrete sequences our abstract compositional ones.

Example 4. Extraction of the abstract sequence from the concrete one in Example 3:

$\langle \texttt{true}, \emptyset, \{B =< C\#4\}, C = 7 \rangle\ \langle C = 7, \emptyset, \{B =< C\#4\}, C = 7 \rangle\ \langle C = 7, \emptyset, \{B =< C\#4\}, C = 7 \rangle$
$\langle C = 7, \emptyset, \{B =< C\#4\}, (A = C \wedge C = 7) \rangle\ \langle (A = C \wedge C = 7), \emptyset, \{B =< C\#4\}, (A = C \wedge C = 7) \rangle$
$\langle (A = C \wedge C = 7), \emptyset, \{B =< C\#4\}, (B = C \wedge A = C \wedge C = 7) \rangle$
$\langle (B = C \wedge A = C \wedge C = 7), \{B =< C\#4\}, \{trs@1, 3\} \rangle$

Then we need the notion of "compatibility" of a tuple w.r.t. a sequence. To this aim we first provide some further notation. Given a sequence of derivation steps

$$\delta = \langle G_1, c_1, T_1, n_1, K_1, G_2, d_1, T_2, n_1' \rangle \ldots \langle G_m, c_m, T_m, n_m, \emptyset, G_m, c_m, T_m, n_m \rangle$$

we denote by $length(\delta)$ and by $instore(\delta)$ the length of the derivation δ and the first input store c_1 respectively. If $t = \langle G_1, c_1, T_1, n_1, K_1, G_2, d_1, T_2, n_1' \rangle$ we define

$V_{loc}(t) = Fv(G_2, d_1) \setminus Fv(G_1, c_1, K_1)$,
$V_{ass}(\delta) = \bigcup_{i=1}^{m-1} Fv(K_i)$ (the variables in the assumptions of δ),
$V_{stable}(\delta) = Fv(G_m)$ (the variables in all the stable sets of δ),
$V_{constr}(\delta) = \bigcup_{i=1}^{m-1} Fv(d_i) \setminus Fv(c_i)$ (the variables in the output constraints of δ which are not in the corresponding input constraints) and
$V_{loc}(\delta) = \bigcup_{i=1}^{m-1} Fv(G_{i+1}, d_i) \setminus Fv(G_i, c_i, K_i)$ (the local variables of δ, namely the variables in the clauses added during the derivation δ and not present in the initial configuration).

We then define the compatibility as follows.

Definition 5 (Compatibility). *[8] Let $t = \langle G_1, c_1, T_1, n_1, K_1, G_2, d_1, T_2, n_1' \rangle$ a tuple representing a derivation step for the goal G_1 and let*

$$\delta = \langle G_2, c_2, T_2, n_2, K_2, G_3, d_2, T_3, n_2' \rangle \ldots \langle G_m, c_m, T_m, n_m, \emptyset, G_m, c_m, T_m, n_m \rangle$$

be a sequence of derivation steps for G_2. We say that t is compatible with δ if the following hold:

1. $CT \models c_2 \rightarrow d_1$,
2. $V_{loc}(\delta) \cap Fv(t) = \emptyset$,
3. $V_{loc}(t) \cap V_{ass}(\delta) = \emptyset$ *and*
4. *for* $i \in [2, m]$, $V_{loc}(t) \cap Fv(c_i) \subseteq \bigcup_{j=1}^{i-1} Fv(d_j) \cup V_{stable}(\delta)$.

The first three conditions reflect the monotonic nature of computations, that the clauses in a derivation are renamed apart and that the variables in the assumptions are disjoint from the variables in the clauses used in a derivation. The last condition ensures that the local variables in a derivation δ and in the abstraction of δ are the same.

Note that if t is compatible with δ then, by using the notation above, $t \cdot \delta$ is a sequence of derivation steps for G_1. We can now define the compositional semantics.

Definition 6 (Compositional semantics). *Let P be a program and let G be a goal. The compositional semantics of G in the program P, $\mathcal{S}_P : Goals \rightarrow \wp(\mathcal{D})$, is defined as*

$$\mathcal{S}_P(G) = \alpha(\mathcal{S}'_P(G))$$

where α is the pointwise extension to sets of the operator given in Definition 4 and $\mathcal{S}'_P : Goals \rightarrow \wp(\mathcal{S}eq)$ is defined as follows:

$$\begin{aligned}\mathcal{S}'_P(G) = \{\langle \tilde{G}_1, c_1, T_1, n_1, \tilde{K}_1, \tilde{G}_2, d_1, T_2, n_1' \rangle \cdot \delta \in \mathcal{S}eq \mid \\ \tilde{G}_1 \text{ is an identified version of } G, \\ CT \not\models c_1 \leftrightarrow \texttt{false},\ \langle \tilde{G}_1, c_1, T_1 \rangle_{n_1} \longrightarrow_P^{K_1} \langle \tilde{G}_2, d_1, T_2 \rangle_{n_1'} \\ \text{and } \delta \in \mathcal{S}'_P(G_2) \text{ for some } \delta \text{ such that} \\ \langle \tilde{G}_1, c_1, T_1, n_1, \tilde{K}_1, \tilde{G}_2, d_1, T_2, n_1' \rangle \text{ is compatible with } \delta\} \cup \\ \{\langle \tilde{G}, c, T, n, \emptyset, \tilde{G}, c, T, n \rangle \in \mathcal{S}eq\}.\end{aligned}$$

$\mathcal{S}'_P(G)$ is also the least fixed-point of the corresponding operator $\Phi \in (Goals \to \wp(\mathcal{S}eq)) \to Goals \to \wp(\mathcal{S}eq)$ defined by

$$\begin{aligned}\Phi(I)(G) = \{\langle \tilde{G}_1, c_1, T_1, n_1, \tilde{K}_1, \tilde{G}_2, d_1, T_2, n'_1\rangle \cdot \delta \in \mathcal{S}eq \mid \\ \tilde{G}_1 \text{ is an identified version of } G, \\ CT \not\models c_1 \leftrightarrow \texttt{false},\ \langle \tilde{G}_1, c_1, T_1\rangle_{n_1} \longrightarrow_P^{K_1} \langle \tilde{G}_2, d_1, T_2\rangle_{n'_1} \\ \text{and } \delta \in I(G_2) \text{ for some } \delta \text{ such that} \\ \langle \tilde{G}_1, c_1, T_1, n_1, \tilde{K}_1, \tilde{G}_2, d_1, T_2, n'_1\rangle \text{ is compatible with } \delta\} \cup \\ \{\langle \tilde{G}, c, T, n, \emptyset, \tilde{G}, c, T, n\rangle \in \mathcal{S}eq\}.\end{aligned}$$

In the above definition, $I : Goals \to \wp(\mathcal{S}eq)$ stands for a generic interpretation assigning to a goal a set of sequences, and the ordering on the set of interpretations $Goals \to \wp(\mathcal{S}eq)$ is that of (point-wise extended) set-inclusion. It is straightforward to check that Φ is continuous (on a CPO), thus standard results ensure that the fixpoint can be calculated by $\sqcup_{n\geq 0}\phi^n(\bot)$, where ϕ^0 is the identity map and for $n > 0$, $\phi^n = \phi \circ \phi^{n-1}$

4 Compositionality and correctness

In this section we prove that the semantics defined above is and-compositional and correct w.r.t. the observables $\mathcal{SA}_P$.

In order to prove the compositionality results, we need to define how to compose two sets of sequences (Definition 10). Let S_1 and S_2 these sets we aim to obtain a set S with the composed sequences. First of all every element $\sigma_1 \in S_1$ is interleaved with each element $\sigma_2 \in S_2$ (Definition 7). Than η operator (Definition 9) is applied to these interleaved sequences: it satisfies the assumptions with the stable atoms of each of these sequences in every possible way and adds to the interleaved sequences the new ones. A substitution is used to satisfy an assumption using a stable atom (Definition 8).

Let $\sigma = \langle c_1, K_1, H_1, d_1\rangle\langle c_2, K_2, H_2, d_2\rangle \cdots \langle c_m, H_m, T_m\rangle \in \mathcal{D}$, we define the overloaded operator $id(\sigma) = id(\bigcup_{i=1}^{m-1} K_i) \cup id(\bigcup_{i=1}^{m} H_i)$ as the set of identification values of all CHR constraints in σ.

Then follows the definition of $\|$ operator for two sequences: every tuple of each sequence is interleaved with the ones of other sequence in all possible ways.

We need a notation for the sequences in $\mathcal{D}$ analogous to that one introduced for sequences of derivation steps to introduce the composition operator $\|$ on abstract sequences. Let $\sigma = \langle c_1, K_1, H_1, d_1\rangle\langle c_2, K_2, H_2, d_2\rangle \cdots \langle c_m, H_m, T_m\rangle \in \mathcal{D}$ be a sequence for the goal G. We define

- $V_{ass}(\sigma) = \bigcup_{i=1}^{m-1} Fv(K_i)$ (the variables in the assumptions of σ),
- $V_{stable}(\sigma) = Fv(H_m) = \bigcup_{i=1}^{m} Fv(H_i)$ (the variables in the stable sets of σ),
- $V_{constr}(\sigma) = \bigcup_{i=1}^{m-1} Fv(d_i) \setminus Fv(c_i)$ (the variables in the output constraints of σ which are not in the corresponding input constraints),
- $V_{loc}(\sigma) = (V_{constr}(\sigma) \cup V_{stable}(\sigma)) \setminus (V_{ass}(\sigma) \cup Fv(G))$ (by using Condition 4 of definition of compatibility the local variables of a sequence σ are the local variables of the derivations δ such $\alpha(\delta) \simeq \sigma$).

Definition 7 (Composition). *The operator* $\|$: $\mathcal{D} \times \mathcal{D} \rightarrow \wp(\mathcal{D})$ *is defined as follows. Let* $\sigma_1, \sigma_2 \in \mathcal{D}$ *be sequences for the goals* H *and* G*, respectively, such that* $id(\sigma_1) \cap id(\sigma_2) = \emptyset$ *and*

$$(V_{loc}(\sigma_1) \cup Fv(H)) \cap (V_{loc}(\sigma_2) \cup Fv(G)) = Fv(H) \cap Fv(G) \qquad (1)$$

then $\sigma_1 \parallel \sigma_2$ *is defined by cases as follows:*

1. *If both* σ_1 *and* σ_2 *have length* 1 *and have the same store, say* $\sigma_1 = \langle c, \tilde{H}, T\rangle$ *and* $\sigma_2 = \langle c, \tilde{G}, T'\rangle$*, then*

$$\sigma_1 \parallel \sigma_2 = \{\langle c, \tilde{H} \cup \tilde{G}, T \cup T'\rangle \in \mathcal{D}\}.$$

2. *If* $\sigma_2 = \langle e, \tilde{G}, T'\rangle$ *has length* 1 *and* $\sigma_1 = \langle c_1, K_1, H_1, d_1\rangle \cdot \sigma'_1$ *has length* > 1 *then*

$$\sigma_1 \parallel \sigma_2 = \{\langle c_1, K_1, H_1 \cup \tilde{G}, d_1\rangle \cdot \sigma \in \mathcal{D} \mid \sigma \in \sigma'_1 \parallel \sigma_2\}.$$

3. *If* $\sigma_1 = \langle c, \tilde{H}, T\rangle$ *has length* 1 *and* $\sigma_2 = \langle e_1, J_1, Y_1, f_1\rangle \cdot \sigma'_2$ *has length* > 1 *then*

$$\sigma_1 \parallel \sigma_2 = \{\langle e_1, J_1, \tilde{H} \cup Y_1, f_1\rangle \cdot \sigma \in \mathcal{D} \mid \sigma \in \sigma_1 \parallel \sigma'_2\}.$$

4. *If both* $\sigma_1 = \langle c_1, K_1, H_1, d_1\rangle \cdot \sigma'_1$ *and* $\sigma_2 = \langle e_1, J_1, Y_1, f_1\rangle \cdot \sigma'_2$ *have length* > 1 *then*

$$\begin{aligned}\sigma_1 \parallel \sigma_2 = &\{\langle c_1, K_1, H_1 \cup Y_1, d_1\rangle \cdot \sigma \in \mathcal{D} \mid \sigma \in \sigma'_1 \parallel \sigma_2\}\\ &\cup\\ &\{\langle e_1, J_1, H_1 \cup Y_1, f_1\rangle \cdot \sigma \in \mathcal{D} \mid \ \sigma \in \sigma_1 \parallel \sigma'_2\}\end{aligned}$$

We underline that the condition (1) in the Definition 7 imposes that there are no common local variables between the two sequences σ_1 and σ_2, so the $Apply'$ rules applied to δ_1 and δ_2, such that $\alpha(\delta_1) = \sigma_1$ and $\alpha(\delta_2) = \sigma_2$, are rules with no common local variables; after that we can observe that the local variables of one sequence are different from the ones in the first goal of the other sequence. Moreover, the condition $id(\sigma_1) \cap id(\sigma_2) = \emptyset$ avoids the capture of identifiers.

Example 5. Computation and abstract sequence of the goals $(A =< B, C =< A)$ and $(C = 7, B =< C, B =< C)$ that is a possible goal subdivision of the one in Example 2.

$$\begin{array}{ll}
d = & \\
\langle\{A =< B\#1, C =< A\#2\}, C = 7, \emptyset\rangle_4 \rightarrow^{\{B=<C\}} & trs@1,5\\
\langle\{A =< C\#6, B =< C\#5, A =< B\#1, C =< A\#2\}, C = 7, \{trs@1,5\}\rangle_6 \rightarrow^{\emptyset} & asx@6,2\\
\langle\{A = C, B =< C\#5, A =< B\#1\}, C = 7, \{trs@1,5\}\rangle_6 \rightarrow^{\emptyset} & Solve\\
\langle\{B =< C\#5, A =< B\#1\}, (A = C \wedge C = 7), \{trs@1,5\}\rangle_6 \rightarrow^{\emptyset} & asx@5,1\\
\langle\{B = C\}, (A = C \wedge C = 7), \{trs@1,5\}\rangle_6 \rightarrow^{\emptyset} & Solve\\
\langle\emptyset, (B = C \wedge A = C \wedge C = 7), \{trs@1,5\}\rangle_6 &
\end{array}$$

Let δ the sequence from computation d, then $\alpha(\delta)$ follows:

$$\begin{array}{ll}
\langle C = 7, \{B =< C\#5\}, \emptyset, C = 7\rangle & (a)\\
\langle C = 7, \emptyset, \emptyset, C = 7\rangle & (b)\\
\langle C = 7, \emptyset, \emptyset, (A = C \wedge C = 7)\rangle & (c)\\
\langle (A = C \wedge C = 7), \emptyset, \emptyset, (A = C \wedge C = 7)\rangle & (d)\\
\langle (A = C \wedge C = 7), \emptyset, \emptyset, (B = C \wedge A = C \wedge C = 7)\rangle & (e)\\
\langle (B = C \wedge A = C \wedge C = 7), \emptyset, \{trs@1,5\}\rangle & (f)
\end{array}$$

Moreover we have the following derivation step for $(C = 7, B =< C, B =< C)$

$$\langle \{C = 7, B =< C\#3, B =< C\#4\}, \mathtt{true}, \emptyset\rangle_4 \rightarrow^{\emptyset} \langle \{B =< C\#3, B =< C\#4\}, C = 7, \emptyset\rangle_4 \ \ Solve$$

and therefore we have that

$$\begin{array}{l}\gamma = \langle\ \{C = 7, B =< C\#3, B =< C\#4\}, \mathtt{true}, \emptyset, 4, \emptyset, \{B =< C\#3, B =< C\#4\}, C = 7, \emptyset, 4\rangle \\ \quad\ \langle\ \{B =< C\#3, B =< C\#4\}, (B = C \wedge A = C \wedge C = 7), \emptyset, 4, \emptyset, \\ \quad\quad \{B =< C\#3, B =< C\#4\}, (B = C \wedge A = C \wedge C = 7), \emptyset, 4\rangle\end{array}$$

is a sequence for $(C = 7, B =< C, B =< C)$. Then $\alpha(\gamma)$ follows:

$$\begin{array}{ll}\langle \mathtt{true}, \emptyset, \{B =< C\#3, B =< C\#4\}, C = 7\rangle & (g) \\ \langle (B = C \wedge A = C \wedge C = 7), \{B =< C\#3, B =< C\#4\}, \emptyset\rangle & (h)\end{array}$$

The following operator is used to satisfy an assumption with a stable atom. The identified stable atom $d\#i$ substitutes each assumption $a\#j$ in the sequence and the identifier i substitutes j in every element of token set, provided that token set cardinality does not decrease.

Definition 8 (Substitution operators). *Let T be a token set, S be a set of identified atoms, $id_1, \ldots, id_n, id'_1, \ldots, id'_n$ be identification values and let $g_1\#id_1, \ldots, g_n\#id_n$, $h_1\#id'_1, \ldots, h_n\#id'_n$ be identified atoms.*
Moreover let $\sigma = \langle c_1, K_1, H_1, d_1\rangle\langle c_2, K_2, H_2, d_2\rangle \cdots \langle c_m, H_m, T_m\rangle \in \mathcal{D}$.

- *$T' = T[id_1/id'_1, \ldots, id_n/id'_n]$ is the token set obtained from T by substituting each occurrence of the identifier id_l with id'_l, for $1 \leq l \leq n$. The operation is defined if T and T' have the same cardinality (namely, there are no elements in T, which collapse when we apply the substitution).*
- *$S[g_1\#id_1/h_1\#id'_1, \ldots, g_n\#id_n/h_n\#id'_n]$ is the set of identified atoms obtained from S by substituting each occurrence of the identified atom $g\#id_l$ with $h_l\#id'_l$, for $1 \leq l \leq n$.*
- *$\sigma' = \sigma\,[g_1\#id_1/h_1\#id'_1, \ldots, g_n\#id_n/h_n\#id'_n]$ is defined only if $T'_m = T_m[id_1/id'_1, \ldots, id_n/id'_n]$ is defined and in this case*

$$\sigma' = \langle c_1, K'_1, H'_1, d_1\rangle\langle c_2, K'_2, H'_2, d_2\rangle \cdots \langle c_m, H'_m, T'_m\rangle \in \mathcal{D},$$

with $1 \leq l \leq m-1, 1 \leq p \leq m$, $K'_l = K_l[g_1\#id_1/h_1\#id'_1, \ldots, g_n\#id_n/h_n\#id'_n]$ and $H'_p = H_p[g_1\#id_1/h_1\#id'_1, \ldots, g_n\#id_n/h_n\#id'_n]$.

The operator introduced in Definition 9 performs the satisfaction of assumption using stable atoms in the composed sequence.

Definition 9 (η operator). *Let W be a set of identified CHR atoms, let σ be a sequence in $\mathcal{D}$ of the form $\langle c_1, K_1, H_1, d_1\rangle \ldots \langle c_m, H_m, T_m\rangle$.We denote by $\sigma \setminus W \in \mathcal{D}$ the sequence $\langle c_1, K_1, H_1 \setminus W, d_1\rangle\langle c_2, K_2, H_2 \setminus W, d_2\rangle \ldots \langle c_m, H_m \setminus W, T_m\rangle$, (where the sets difference $H_j \setminus W$ considers identifications, with $1 \leq j \leq m$).*
The operator $\eta : \wp(\mathcal{D}) \rightarrow \wp(\mathcal{D})$ is defined as follows. Given $S \in \wp(\mathcal{D})$, $\eta(S)$ is the least set satisfying the following conditions:

- *$S \subseteq \eta(S)$;*

– *if* $\sigma' \cdot \langle c, K, H, d\rangle \cdot \sigma'' \in \eta(S)$ *and there exist two sets of identifi ed atoms* $K' = \{g_1\#id_1, \ldots, g_n\#id_n\} \subseteq K$ *and* $W = \{h_1\#id'_1, \ldots, h_n\#id'_n\} \subseteq H$ *such that*
  1. *for* $1 \leq l \leq n$, $CT \models (c \wedge g_l) \leftrightarrow (c \wedge h_l)$ *and*
  2. $\bar{\sigma} = ((\sigma' \cdot \langle c, K \backslash K', H, d\rangle \cdot \sigma'') \backslash W)\,[g_1\#id_1/h_1\#id'_1, \ldots, g_n\#id_n/h_n\#id'_n]$ *is defi ned,*

  then $\bar{\sigma} \in \eta(S)$.

Particularly Defi nition 9 introduces an upper closure operator[3] which saturates a set of sequences S by adding new sequences where redundant assumptions can be removed: an assumption $a\#i$ in K can be removed if $a'\#j$ appears as a stable atom in H and the built-in store c implies that a is equivalent to a'. Once a stable atom is "consumed" for satisfying an assumption it is removed from (the sets of stable atoms of) all the tuples appearing in the sequence, to avoid multiple uses of the same atom.

We can now defi ne the composition operator $\|$ on set of sequences. To simplify the notation we denote by $\|$ both the operator acting on sequences and that one acting on sets of sequences. It uses Defi nition 7 to compose the sequences and Defi nition 9 to simplify stable atoms with assumptions.

Defi nition 10 (Set of sequence composition). *The composition of sets of sequences* $\|\colon \wp(\mathcal{D}) \times \wp(\mathcal{D}) \to \wp(\mathcal{D})$ *is defi ned by:*

$$
\begin{aligned}
S_1 \parallel S_2 = \{\sigma \in \mathcal{D} \mid\ & \textit{there exist } \sigma_1 \in S_1 \textit{ and } \sigma_2 \in S_2 \textit{ such that} \\
& \sigma = \langle c_1, K_1, H_1, d_1\rangle \cdots \langle c_m, H_m, T_m\rangle \in \eta(\sigma_1 \parallel \sigma_2), \\
& (V_{loc}(\sigma_1) \cup V_{loc}(\sigma_2)) \cap V_{ass}(\sigma) = \emptyset \textit{ and for } i \in [1, m] \\
& (V_{loc}(\sigma_1) \cup V_{loc}(\sigma_2)) \cap Fv(c_i) \subseteq \textstyle\bigcup_{j=1}^{i-1} Fv(d_j) \cup Fv(H_m)\}.
\end{aligned}
$$

The fi rst condition warrants that local variables of σ, that are the ones used in the derivation of which σ is abstraction, are different from the ones used by assumptions of σ. The second condition warrants that σ is an abstraction of a derivation that enjoys of condition 4 of Defi nition 5 (Compatibility).

Example 6. Application of Defi nition 10 using the two abstract sequences of Example 5. We prove that this composition gives us the one in Example 4. First of all we compose the abstract sequences $\alpha(\delta)$ and $\alpha(\gamma)$ of Example 5 using Defi nition 7: the following is only one of the possible compound abstract sequences that we can make following the Defi nition 7 itself.

$$
\begin{array}{ll}
\langle \mathtt{true}, \emptyset, \{B =< C\#3, B =< C\#4\}, C = 7\rangle & g(a) \\
\langle C = 7, \{B =< C\#5\}, \{B =< C\#3, B =< C\#4\}, C = 7\rangle & a(h) \\
\langle C = 7, \emptyset, \{B =< C\#3, B =< C\#4\}, C = 7\rangle & b(h) \\
\langle C = 7, \emptyset, \{B =< C\#3, B =< C\#4\}, (A = C, C = 7)\rangle & c(h) \\
\langle (A = C, C = 7), \emptyset, \{B =< C\#3, B =< C\#4\}, (A = C, C = 7)\rangle & d(h) \\
\langle (A = C, C = 7), \emptyset, \{B =< C\#3, B =< C\#4\}, (B = C, A = C, C = 7)\rangle & e(h) \\
\langle (B = C, A = C, C = 7), \{B =< C\#3, B =< C\#4\}, \{trs@1, 5\}\rangle & f \text{ and } h
\end{array}
$$

where $g(a)$ means that is used the tuple g and the stable atoms of tuple (a), and so on until the last step of interleaving, f and h, closes the composition. The application

[3] $S \subseteq \eta(S)$ holds by defi nition, and it is easy to see that $\eta(\eta(S)) = \eta(S)$ holds and that $S \subseteq S'$ implies $\eta(S) \subseteq \eta(S')$.

of Definition 9, using Definition 8, substitute the constraint labeled with identifier $\#5$ with the one labeled with $\#3$.

$$
\begin{array}{ll}
\langle \texttt{true}, \emptyset, \{\cancel{B =< C\#2}, B =< C\#4\}, C = 7\rangle & g(a) \\
\langle C = 7, \{\cancel{B =< C\#\ 5\ \rightarrow\ 3}\}, \{\cancel{B =< C\#3}, B =< C\#4\}, C = 7\rangle & a(h) \\
\langle C = 7, \emptyset, \{\cancel{B =< C\#3}, B =< C\#4\}, C = 7\rangle & b(h) \\
\langle C = 7, \emptyset, \{\cancel{B =< C\#3}, B =< C\#4\}, (A = C, C = 7)\rangle & c(h) \\
\langle (A = C, C = 7), \emptyset, \{\cancel{B =< C\#3}, B =< C\#4\}, (A = C, C = 7)\rangle & d(h) \\
\langle (A = C, C = 7), \emptyset, \{\cancel{B =< C\#3}, B =< C\#4\}, (B = C, A = C, C = 7)\rangle & e(h) \\
\langle (B = C, A = C, C = 7), \{\cancel{B =< C\#3}, B =< C\#4\}, \{trs@1,\ \cancel{5} \rightarrow 3\}\rangle & f \text{ and } h
\end{array}
$$

Note that another application of Definition 9 is possible by satisfying the assumption with the stable atom $B =< C\#4$. Definition 9 adds to its set S also the following:

$$
\begin{array}{ll}
\langle \texttt{true}, \emptyset, \{B =< C\#4[3]\}, C = 7\rangle & g(a) \\
\langle C = 7, \emptyset, \{B =< C\#4[3]\}, C = 7\rangle & a(h) \\
\langle C = 7, \emptyset, \{B =< C\#4[3]\}, C = 7\rangle & b(h) \\
\langle C = 7, \emptyset, \{B =< C\#4[3]\}, (A = C, C = 7)\rangle & c(h) \\
\langle (A = C, C = 7), \emptyset, \{B =< C\#4[3]\}, (A = C, C = 7)\rangle & d(h) \\
\langle (A = C, C = 7), \emptyset, \{B =< C\#4[3]\}, (B = C, A = C, C = 7)\rangle & e(h) \\
\langle (B = C, A = C, C = 7), \{B =< C\#4[3]\}, \{trs@1, 3[4]\}\rangle & f \text{ and } h
\end{array}
$$

that is equal to the one of Example 4, where $4[3]$ represents the two possibility: when you satisfy the constraint labeled with $\#5$ with the ones labeled with $\#3$ or $\#4$ respectively.

Using this notion of composition of sequences we can show that the semantics $\mathcal{S}_P$ is compositional. The proof of the following theorem, as well as some technical lemmata used in such a proof, are here omitted for space reasons. We need the following definition before the compositionality theorem statement:

Definition 11. *Let $\sigma, \sigma_1, \sigma_2 \in \mathcal{D}$ and let $S_1, S_2 \in \wp(\mathcal{D})$.*

- *Let $i_1, i_2, \ldots, i_m$ be a permutation of $1, 2, \ldots, m$. $\sigma[1/i_1, \ldots, m/i_m]$ is the sequence obtained from σ, by substituting each occurrence of the identification value j with the corresponding i_j, for $j \in [1, m]$.*
- *$\sigma_1 \simeq \sigma_2$ if there exists a natural number m and there exists a permutation $i_1, i_2, \ldots, i_m$ of $1, 2, \ldots, m$ such that $\sigma_1 = \sigma_2[1/i_1, \ldots, m/i_m]$.*
- *$S_1 \ll S_2$ if for each $\sigma_1 \in S_1$ there exists $\sigma_2 \in S_2$ such that $\sigma_1 \simeq \sigma_2$.*
- *$S_1 \simeq S_2$ if $S_1 \ll S_2$ and $S_2 \ll S_1$.*

Theorem 1 (Compositionality). *Let P be a program and let H and G be two goals. Then $\mathcal{S}_P(H, G) \simeq \mathcal{S}_P(H) \parallel \mathcal{S}_P(G)$.*

4.1 Correctness

In order to show the correctness of the semantics $\mathcal{S}_P$ w.r.t. the (input/output) observables $\mathcal{SA}_P$, we first introduce a different characterization of $\mathcal{SA}_P$ obtained by using the new transition system defined in Table 2.

Definition 12. *Let P be a program and let G be a goal and let $\longrightarrow_P$ be (the least relation) defined by the rules in Table 2. We define*

$$\mathcal{SA}'_P(G) = \{\exists_{-Fv(G)} c \mid \langle G, \emptyset, \emptyset\rangle_{n_1} \longrightarrow_P^{\emptyset} \cdots \longrightarrow_P^{\emptyset} \langle \emptyset, c, T_m\rangle_{n_m} \not\longrightarrow_P^{K}\}.$$

The correspondence of $\mathcal{SA}'$ with the original notion $\mathcal{SA}$ is stated by the following proposition, whose proof is immediate.

Proposition 1. *Let P be a program and let G be a goal. Then $\mathcal{SA}_P(G) = \mathcal{SA}'_P(G)$.*

The observables $\mathcal{SA}'_P$, and therefore $\mathcal{SA}_P$, describing answers of successful computations can be obtained from $\mathcal{S}_P$ by considering suitable sequences, namely those sequences which do not perform assumptions neither on CHR constraints nor on built-in constraints. The first condition means that the second component of tuples of the sequence of our compositional semantic ($\langle c, K, H, d\rangle$) must be empty, while the second one means that the assumed constraint at step i must be equal to the produced constraint of steps $i-1$. We call "connected" those sequences which satisfy these requirements:

Definition 13 (Connected sequences). *Assume that*

$$\sigma = \langle c_1, K_1, H_1, d_1\rangle\langle c_2, K_2, H_2, d_2\rangle \ldots \langle c_m, H_m, T_m\rangle$$

is a sequence in $\mathcal{D}$. We say that σ is connected if $K_j = \emptyset$ for each j, $1 \leq j \leq m-1$ and $d_j = c_{j+1}$.

The proof of the following result derives from the definition of connected sequence and an easy inductive argument. If $\sigma = \langle c_1, K_1, H_1, d_1\rangle \ldots \langle c_m, H_m, T_m\rangle$ is a sequence, we denote by $store(\sigma)$ the built-in constraint c_m and by $lastg(\sigma)$ the goal H_m.

Proposition 2. *Let P be a program and let G be a goal. Then*

$$\mathcal{SA}'_P(G) = \{\exists_{-Fv(G)} c \mid \text{there exists } \sigma \in \mathcal{S}_P(G) \text{ such that } instore(\sigma) = \emptyset, \\ \sigma \text{ is connected, } lastg(\sigma) = \emptyset \text{ and } c = store(\sigma)\}.$$

The following corollary is immediate from Proposition 1.

Corollary 1 (Correctness). *Let P be a program and let G be a goal. Then*

$$\mathcal{SA}_P(G) = \{\exists_{-Fv(G)} c \mid \text{there exists } \sigma \in \mathcal{S}_P(G) \text{ such that } instore(\sigma) = \emptyset, \\ \sigma \text{ is connected, } lastg(\sigma) = \emptyset \text{ and } c = store(\sigma)\}.$$

5 Conclusions

In this paper we have introduced a semantics for CHR which is compositional w.r.t. the and-composition, which is correct w.r.t "success answers" and which takes into account the token store used in the refined operational semantics to avoid trivial non termination due to the propagation rule. This work can then be seen as an extension of [8] and could be further extended along several different lines.

Firstly, as already mentioned in [8], it would be desirable to obtain a compositional characterization also for those answers obtained by considering computations terminating with a user-defined constraint which does not need to be empty.

A second possible extension is the investigation of the full abstraction issue. For obvious reasons it would be desirable to introduce in the semantics the minimum amount of information needed to obtain compositionality, while preserving correctness. Such a full abstraction result seems difficult to achieve, however techniques similar to those used in [5, 3] for analogous results in the context of ccp could be considered.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In G. Smolka. editor, Proc. *Third Int'l Conf. on Principles and Practice of Constraint Programming (CP 97)*, Lecture Notes in Computer Science 1330. Springer-Verlag, 1997.
2. S. Abdennadher, E. Krämer, M. Saft, and M. Schmauss. JACK: a java constraint kit. *Electronic Notes in Theoretical Computer Science*, 64. Elsevier, 2000.
3. F.S. de Boer, M. Gabbrielli, and M.C. Meo. Semantics and expressive power of a timed concurrent constraint language. In G. Smolka. editor, Proc. *Third Int'l Conf. on Principles and Practice of Constraint Programming (CP 97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.
4. F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures in a paradigm for asynchronous communication. In J.C.M. Baeten and J.F. Groote, editors, *Proceedings of CONCUR'91*, vol. 527 of *LNCS*, pages 111–126. Springer-Verlag, 1991.
5. F.S. de Boer and C. Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of TAPSOFT/CAAP*, vol. 493 of *LNCS*, pages 296–319. Springer-Verlag, 1991.
6. A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. *Theoretical Computer Science* 122(1-2): 3–47, 1994.
7. S. Brookes. A fully abstract semantics of a shared variable parallel language. In *Proc. Eighth IEEE Symposium on Logic In Computer Science*. IEEE Computer Society Press, 1993.
8. G. Delzanno, M. Gabbrielli, and M. C. Meo. A Compositional Semantics for CHR In *PPDP'05* Lisbon, Portugal. Copyright 2005 ACM.
9. G.J. Duck, P.J. Stuckey, M.G. de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *ICLP 2004*, pages 90–104. Springer, 2004.
10. T. Frühwirth. Introducing simplification rules. TR ECRC-LP-63, ECRC Munich. October 1991.
11. T. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37 (1-3):95-138, 1998.
12. M. Garcia de la Banda, B. Demoen, K. Mariott, and P. J. Stuckey. To the gates of HAL: a HAL tutorial. In *Proceedings of the International Symposium on Functional and Logic Programming - FLOPS 2002*, pages 47 – 66, Japan, 2002. LNCS.
13. C. Holzbaur, and T. Früwirth. A prolog constraint handling rules compiler and runtime system. *Journal of Applied Artificial Intelligence*, 14(4), 2000.
14. B. Jonsson. A model and a proof system for asynchronous processes. In *Proc. of the 4th ACM Symp. on Principles of Distributed Computing*, pages 49–58. ACM Press, 1985.
15. T. Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, Katholieke Universiteit Leuven, Celestinjneanlaan 200 A - B - 3001 Leuven, June 2005.
16. V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of POPL*, pages 232–245. ACM Press, 1990.

Search Strategies in CHR(Prolog)

Leslie De Koninck*, Tom Schrijvers**, Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium
{leslie,toms,bmd}@cs.kuleuven.be

Abstract We extend the refined operational semantics of the Constraint Handling Rules language to support the implementation of different search strategies. Such search strategies are necessary to build efficient Constraint Logic Programming systems. This semantics is then further refined so that it is more suitable as a basis for a trailing based implementation. We propose a source to source transformation to implement breadth first search in CHR(Prolog): CHR with Prolog as the host language. Breadth first is chosen because it exhibits the main difficulties in the implementation of search strategies, while being easy to understand. We evaluate our implementation and give directions for future work.

1 Introduction

"Algorithm = Logic + Control" is a famous quote by Robert Kowalski, implying a separation between the declarative meaning of a program and its operational behavior. The latter consists of choosing the order in which different conjunctives and disjunctives are processed. In the context of Constraint Logic Programming, the order of the conjunctives is handled by scheduling and the order of disjunctives by search. In this paper we focus on the latter.

Search strategy is considered to be a crucial component in making CLP systems efficient. The standard left-to-right depth first search which is often given to a CLP system by its underlying Prolog implementation, does not always lead to the best results. In this paper, we investigate how search strategies can be implemented in the Constraint Handling Rules (CHR) language.

CHR [8] is a high-level rule-based language, built on top of a host language like Prolog [15], Java [2], Haskell or Curry [9], and designed for a more easy implementation of Constraint Programming facilities. CHR can be extended to CHR$^\vee$ by allowing disjunctions in the rule bodies [3]. This extension makes it possible to perform search in CHR. Implementations of CHR(Prolog): CHR on top of Prolog already are implementations of CHR$^\vee$ as well. These implementations handle disjunctions by using the Prolog built-in (depth first) search mechanism.

Our aim is to implement different search strategies in CHR(Prolog). We start with a more detailed look at the context, motivation and goals of this paper

* Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)
** Research Assistant of the Research Foundation - Flanders (F.W.O.-Vlaanderen).

in Section 2. In Section 3, we extend the refined operational semantics of CHR towards $CHR^{\vee}$, supporting the definition of different search strategies. In Section 4, we further refine this semantics to make it more suitable for a trailing based implementation. Then, in Section 5, we propose a source to source transformation to implement breadth first search, making use of the K.U.Leuven CHR system [15] on top of SWI-Prolog [20]. We evaluate our approach in Section 6 and give an overview of related work in Section 7. Section 8 concludes.

2 Context, Motivation and Goals

We investigate how to implement different search strategies in the K.U.Leuven CHR system, running on top of SWI-Prolog. The main motivation is that we are developing a Constraint Logic Programming system, capable of handling nonlinear constraints over the real numbers, called INCLP($\mathbb{R}$) [13]. This system is implemented using the above mentioned CHR and Prolog implementations.

Search strategies are a fundamental part of CLP systems. A well chosen strategy can decrease the runtime considerably. Although CHR was originally designed for the implementation of constraint solvers, its limited ability to implement search algorithms, especially for CHR(Prolog) implementations, is a major weakness. We note that this limitation is mainly caused by the host language. Languages like Java that do not have a built-in search mechanism, need an explicit implementation of search and often offer more freedom with respect to search strategies. For example, the Java Constraint Kit [2] offers different search strategies by using its Java Abstract Search Engine module [14].

2.1 Breadth First Search

In this paper, we focus on the breadth first search strategy. This is a very basic strategy, but it exhibits two important challenges in implementing search strategies. Here, we take a closer look at these problems. Figure 1 depicts a search tree that is traversed in breadth first order. The nodes are visited in alphabetical order and the edges in numerical order. The edges 3 and 6 are edges that have been processed before, but need to be reapplied in order to be able to reach the next unvisited node. If this is done by recomputing them from scratch, we are essentially performing iterative deepening instead of breadth first search.

Non-standard Visiting Order When a choice point is created in Prolog, all its alternatives have to be tried before backtracking to a higher level. When visiting a node in breadth first order, the children of the current node are alternatives that are to be postponed until after all nodes at the current level are traversed. To implement such an order, we need a global representation of the choice points.

Revisiting Nodes Another difficulty is that in breadth first search, nodes of the search tree are revisited when going to the next level. It is important to note here that we do not want to repeat all the computations on the edge that terminated

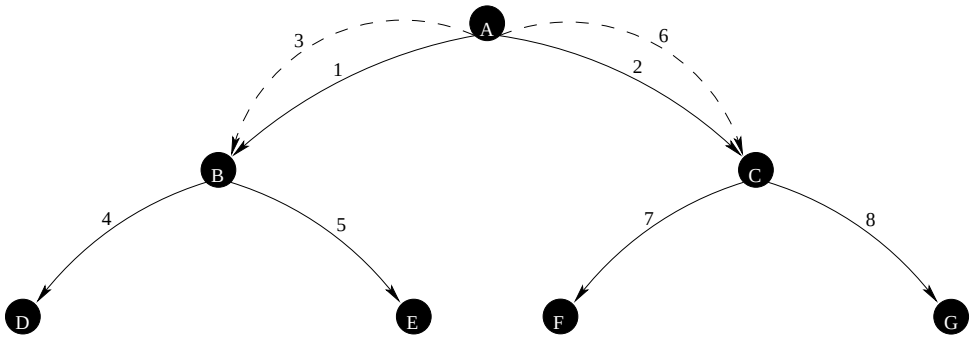


Figure1. Search Tree

in the repeated node. Instead, we wish to store the result and load this result when revisiting. In the context of CLP, nodes correspond to splitting the domain of a variable and the edges that connect the nodes represent the computations to return to a consistent state. Reaching a consistent state often takes a lot of time to compute, but quite limited space to store the result. For example, the INCLP($\mathbb{R}$) system uses a computationally quite expensive interval Newton iteration to achieve consistency, whereas the result only consists of some changed variable domains. Note that the ability to quickly move from one node to another already visited node, forms the main difference between breadth first search and iterative deepening.

2.2 Goals

The goals of this paper are the following:

- to define an extension of the refined operational semantics that supports disjunctions in the rule bodies and allows to define different search strategies
- to prove that it is possible to implement a search strategy different from depth first, by only using the language features of CHR and Prolog that are currently available
- to investigate what kind of extra support from the host language and the CHR system could make our task much easier and/or offer a better performance

3 A Combination of the Refined Operational Semantics ω_r and CHR$^\vee$

In this section, we briefly review the theoretical operational semantics ω_t [1] of CHR and how it is extended to become a semantics for CHR$^\vee$ [3]. We then review the refined operational semantics ω_r [6] and show how it can be extended in a similar way.

3.1 The Theoretical Operational Semantics ω_t and CHR$^\vee$

A CHR constraint c is an atom $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms. An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with a unique identifier i. We introduce the functions *chr* and *id*, defined as $chr(c\#i) = c$ and $id(c\#i) = i$. A CHR execution state is denoted by a tuple $\langle G, S, B, T\rangle_n$ where G is the *goal*, S the *CHR constraint store*, B the *built-in store*, T the *propagation history* and n the *next free identifier*. We assume familiarity with these concepts, see for example [6].

The operational semantics is described by a list of transitions that transform one CHR execution state into another one. If no more transitions are applicable, a *final* execution state has been reached. This is either a state with an empty goal: a successful final execution state, or a state with an inconsistent built-in store: a failed final execution state. The transitions of ω_t can be found in [6].

CHR$^\vee$ is an extension of CHR where disjunctions are allowed in the bodies of rules. A theoretical semantics $\omega_t^\vee$ for CHR$^\vee$ is presented as an extension of ω_t. We introduce a set of CHR execution states called the set of alternatives: $\bar{E} = \{E_1, \ldots, E_n\}$. The following transitions manipulate the set of alternatives and are an extension of [3]:

S1. Derive $\{\sigma\} \cup \bar{E} \rightarrowtail \{\sigma'\} \cup \bar{E}$ if there exists a transition $\sigma \rightarrowtail_{\omega_t} \sigma'$.
S2. Split $\{\langle (G_1 \vee G_2) \wedge G, S, B, T\rangle_n\} \cup \bar{E} \rightarrowtail \{\langle G_1 \wedge G, S, B, T\rangle_n, \langle G_2 \wedge G, S, B, T\rangle_n\} \cup \bar{E}$.

The **Derive** transition supports both processing states in parallel or sequentially in any order. It is desirable to have more control over the search order, but since the theoretical operational semantics is already highly nondeterministic, it is not very meaningful to impose a particular search order on it.

3.2 The Refined Operational Semantics ω_r

The refined operational semantics ω_r is an implementation of the theoretical operational semantics that makes the execution considerably more deterministic and is more closely related to actual implementations of CHR.

A CHR execution state is a tuple $\langle A, S, B, T\rangle_n$ where A is the *execution stack* and S, B, T and n are as in ω_t. The transitions of ω_r can be found in [6].

3.3 A Refined Operational Semantics for CHR$^\vee$

In this subsection, we introduce the $\omega_r^\vee$ semantics for CHR$^\vee$ which extends the refined operational semantics of CHR. This extension supports the definition of search strategies, which is not present in the theoretical semantics $\omega_t^\vee$.

We make the set of alternatives of $\omega_t^\vee$ more concrete by implementing it as an ordered sequence of CHR execution states called the list of alternatives: $\bar{E} = [E_1, E_2, \ldots, E_n]$ where the state E_1 is the active execution state and $E_2, \ldots, E_n$ are the remaining alternatives. We propose the following transitions that manipulate the list of alternatives:

S1. Derive $[\sigma \mid \bar{E}] \rightarrowtail [\sigma' \mid \bar{E}]$ if there exists a transition $\sigma \rightarrowtail_{\omega_r} \sigma'$.

S2a. Split (Depth First) $[\sigma \mid \bar{E}] \rightarrowtail [\sigma_1, \ldots, \sigma_m \mid \bar{E}]$ where $\sigma = \langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T \rangle_n$ and $\sigma_i = \langle [A_i \mid A], S, B, T \rangle_n$ for $1 \leq i \leq m$. This transition implements a depth first search.

S2b. Split (Breadth First) $[\sigma \mid \bar{E}] \rightarrowtail \bar{E} +\!\!+ [\sigma_1, \ldots, \sigma_m]$ where $\sigma = \langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T \rangle_n$ and $\sigma_i = \langle [A_i \mid A], S, B, T \rangle_n$ for $1 \leq i \leq m$. This transition implements a breadth first search.

S3. Next $[\sigma \mid \bar{E}] \rightarrowtail \bar{E}$ if σ is final CHR execution state. This transition is applied automatically if E is a failed final state, but requires an external event (explicit call for the next solution, e.g. from the toplevel) if E is a successful final state.

The given **Split** transitions are only two of the possibilities. Other search strategies can be implemented easily using similar transitions. For example, best first search can be implemented by sorting the list of remaining alternatives according to some heuristic. Strategies like intelligent backtracking can be implemented by removing states from the remaining alternatives. A branch and bound algorithm can be implemented by adding the provisional optimum as a constraint to the remaining alternatives.

We require that at the moment of a **Next** transition, it is known which of the alternatives is to be chosen next. This implies that it is not always necessary to have a full order between the different alternatives and allows implementing more dynamic search strategies.

While the **Split** transition in $\omega_t^\vee$ only supports binary disjunctions, the $\omega_r^\vee$ **Split** transitions supports n-ary disjunctions. The reason for this generalization is that although n-ary disjunctions can be logically modeled as a series of nested binary disjunctions, they do not behave equivalently with respect to all search strategies.

4 The Tree-based Operational Semantics $\omega_\curlywedge$

In the above presentation, the **Split** transitions create copies of the CHR constraint store, the built-in store and the propagation history for each of the alternatives. These copies can then be changed independently by the ω_r transitions. In practice copying the execution state is often very expensive and should be avoided if possible. It is also hard to implement in our current CHR implementation. Another issue is that when changing the active execution state by using the **Split** or **Next** transitions, we have ignored the fact that these execution states are often largely the same. These considerations influence implementations of the proposed semantics.

In this section, we propose the tree-based operational semantics $\omega_\curlywedge$[1] which is a refinement of the $\omega_r^\vee$ semantics. It is based on the concept of a search tree and is more suitable as a basis for a practical implementation based on trailing. It makes the presentation in Section 5 more straightforward.

[1] Read as "omega tree".

4.1 Nodes and Edges

A search tree consists of a set of nodes and a set of (directed) edges connecting these nodes. A node is either an *internal* node or a *leaf* node. An internal node represents a choice point; the *root* node is a special internal node that starts the whole search process. A leaf node represents a successful or failed final execution state. An edge goes from one node, its *start* node, to another node, its *end* node, and represents the derivation that transforms one of the alternatives of its start node into its end node. Edges come in two flavors: *processed* edges and *unprocessed* edges. For a processed edge the derivation from start to end node has already been computed, and for unprocessed edges it has not yet. We now introduce a mapping between these concepts and CHR execution states.

Nodes are represented as CHR execution states. An internal node is represented by the state $\langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T \rangle_n$ where $m \geq 2$. The root node of the search tree is the initial state $\langle G, \emptyset, true, \emptyset \rangle_1$ with G the initial goal. A leaf node is simply a final state.

An unprocessed edge is represented by the CHR execution state $N = \langle [A_i \mid A], S, B, T \rangle_n$. The edge is processed by starting a derivation in that state. Such a derivation ends either in a successful or failed final CHR execution state (a leaf node) or in another choice point (an internal node). A processed edge connects its start node N with its end node N' and is represented as $N \frown N'$.

A search tree state is a tuple $\langle \sigma, N, \bar{E}_u, \bar{E}_p \rangle$. Here, σ is the current CHR execution state of the active edge or $\sigma = \epsilon$ if no edge is active. N is the current start node, which is also the start node of the active edge if an edge is active. $\bar{E}_u$ is an ordered sequence of (inactive) unprocessed edges and $\bar{E}_p$ is a set of processed edges. The edges in $\bar{E}_p$ form the part of the search tree that has already been explored and the edges in $\bar{E}_u$ form the boundary between the explored part and the unexplored part of the search tree. The initial search tree state is the tuple $\langle \langle G, \emptyset, true, \emptyset \rangle_1, \langle G, \emptyset, true, \emptyset \rangle_1, \epsilon, \emptyset \rangle$ The following transitions manipulate search tree states:

S1. Derive $\langle \sigma, N, \bar{E}_u, \bar{E}_p \rangle \rightarrowtail \langle \sigma', N, \bar{E}_u, \bar{E}_p \rangle$ if there is a transition $\sigma \rightarrowtail_{\omega_r} \sigma'$.

S2a. Split (Depth First) $\langle \sigma, N, \bar{E}_u, \bar{E}_p \rangle \rightarrowtail \langle \epsilon, \sigma, [\sigma_1, \ldots, \sigma_m \mid \bar{E}_u], \bar{E}_p \cup \{N \frown \sigma\} \rangle$ where $\sigma = \langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T \rangle_n$ and $\sigma_i = \langle [A_i \mid A], S, B, T \rangle_n$ for $i \in 1 \ldots m$. This transition implements depth first search. The choice point creates a new node σ which becomes the current start node. The edge $N \frown \sigma$ is added to the set of processed edges. For every alternative of the choice point, a new unprocessed edge, starting in the node σ, is added in front of the list of unprocessed edges.

S2b. Split (Breadth First) $\langle \sigma, N, \bar{E}_u, \bar{E}_p \rangle \rightarrowtail \langle \epsilon, \sigma, \bar{E}_u +\!\!+ [\sigma_1, \ldots, \sigma_m \mid \bar{E}_u], \bar{E}_p \cup \{N \frown \sigma\} \rangle$ where $\sigma = \langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T \rangle_n$ and $\sigma_i = \langle [A_i \mid A], S, B, T \rangle_n$ for $i \in 1 \ldots m$. This transition implements breadth first search. The choice point creates a new node σ which becomes the current start node. The edge $N \frown \sigma$ is added to the set of processed edges. For every alternative of the choice point, a new unprocessed edge, starting in the node σ, is added to the back of the list of unprocessed edges.

S3. Next $\langle \epsilon, N, [\sigma_i \mid \bar{E}_u], \bar{E}_p \rangle \rightarrowtail \langle \sigma_i, N, \bar{E}_u, \bar{E}_p \rangle$ where $N = \langle [A_1 \vee \ldots \vee A_m \mid A], S, B, T \rangle_n$ and $\sigma_i = \langle [A_i \mid A], S, B, T \rangle_n$. This transition activates the next unprocessed edge. It requires that the current start node corresponds to the start node of the next unprocessed edge.

S4. Move Down $\langle \epsilon, N, [\sigma_i \mid \bar{E}_u], \bar{E}_p \cup \{N \frown N'\} \rangle \rightarrowtail \langle \epsilon, N', [\sigma_i \mid \bar{E}_u], \bar{E}_p \cup \{N \frown N'\} \rangle$ if node N is an ancestor of N'', the start node of σ_i, and the edge $N \frown N'$ is on the path between nodes N and N'' or $N' = N''$

S5a. Move Up (Internal Node) $\langle \epsilon, N, [\sigma_i \mid \bar{E}_u], \bar{E}_p \cup \{N' \frown N\} \rangle \rightarrowtail \langle \epsilon, N', [\sigma_i \mid \bar{E}_u], \bar{E}_p \cup \{N' \frown N\} \rangle$ if node N is not an ancestor node of the start node of σ_i.

S5b. Move Up (Successful Leaf Node) $\langle \sigma, N, \bar{E}_u, \bar{E}_p \rangle \rightarrowtail \langle \epsilon, N, \bar{E}_u, \bar{E}_p \rangle$ if σ is a successful final state, a solution. To be applied, this transition requires an external event asking for the next solution.

S5c. Move Up (Failed Leaf Node) $\langle \sigma, N, \bar{E}_u, \bar{E}_p \rangle \rightarrowtail \langle \epsilon, N, \bar{E}_u, \bar{E}_p \rangle$ if the execution state σ is a failed final CHR execution state.

4.2 Equivalence of the ω_λ and $\omega_r^\vee$ semantics

We can show that the ω_λ semantics presented in this section, is operationally equivalent to the $\omega_r^\vee$ semantics presented in Section 3. For this purpose, we introduce the following abstraction function:

Definition 1. *We define an abstraction function α that maps a search tree state on a list of alternatives as follows:*

$$\alpha(\langle \langle A, S, B, T \rangle_n, \_, \bar{E}_u, \_ \rangle) = [\langle A, S, B, T \rangle_n \mid \bar{E}_u]$$
$$\alpha(\langle \epsilon, \_, \bar{E}_u, \_ \rangle) = \bar{E}_u$$

The equivalence proof consists of the following theorems:

Theorem 1. *For every two lists of alternatives $\bar{E}_1$ and $\bar{E}_2$ for which holds that $\bar{E}_1 \rightarrowtail_{\omega_r^\vee} \bar{E}_2$, there exist two search tree states S_1 and S_2 for which a derivation $S_1 \rightarrowtail^*_{\omega_\lambda} S_2$ exists such that $\alpha(S_1) = \bar{E}_1$ and $\alpha(S_2) = \bar{E}_2$.*

Theorem 2. *For every two search tree states S_1 and S_2 for which holds that $S_1 \rightarrowtail_{\omega_\lambda} S_2$, we have that either $\alpha(S_1) = \alpha(S_2)$ or there exist two lists of alternatives $\bar{E}_1$ and $\bar{E}_2$ for which a transition $\bar{E}_1 \rightarrowtail_{\omega_r^\vee} \bar{E}_2$ exists such that $\alpha(S_1) = \bar{E}_1$ and $\alpha(S_2) = \bar{E}_2$.*

The proofs of these theorems can be found in [12].

5 Implementation

In this section we give an overview of how a breadth first search strategy can be implemented for the K.U.Leuven CHR system [15] in SWI-Prolog [20] using a source to source transformation. Apart from some practical issues that are implementation specific, most of the ideas that are presented here can be generalized to other CHR and Prolog implementations and even to CHR implementations not running on top of Prolog.

5.1 Nodes and Edges

In Section 4 we have introduced the concepts of nodes and edges. We now present how we can implement the processing of edges. The next unprocessed edge can only become active if the current start node is also the start node of the next edge. If the current start node is a different node, it first needs to be changed: if it is an ancestor of the next edge's start node, there exists an already processed edge on the path between the current start node and the next edge's start node.[2] The changes made by the processed edge are loaded and its end node becomes the current start node. This process is repeated until the current start node is equal to the start node of the next unprocessed edge. Otherwise, if the current start node is not an ancestor of the next edge's start node, the changes on the edge of which the current start node is the end node, are backtracked and the current start node is changed to that edge's start node. This process is repeated until the current start node is an ancestor of the next edge's start node.

For the implementation it is advantageous to also allow already processed edges to become active. This way, we can treat processed and unprocessed edges in a similar way. Instead of initiating a derivation, the activation of a processed edge only causes the current start node to be changed to its end node. When transferring to the next edge in the list of unprocessed edges, processed edges that are on the 'path' between the current start node and the next unprocessed edge's start node, temporarily become active so as to change the current start node.

```
activate_next_edge, active_edge(Edge) <=>
        next_edge(NextEdge),
        same_branch(Edge,NextEdge),
        (   edge_end_node(Edge,Node),
            edge_start_node(NextEdge,Node)
        ->  set_next_edge,
            active_edge(NextEdge)
        ;   next_on_path(Edge,NextEdge,Between),
            active_edge(Between)
        ).
```

The `next_edge/1` predicate unifies the next scheduled edge with its argument. The `same_branch/2` predicate checks whether the two given edges are on a single branch in the search tree. The combination of both predicates must be tried again on backtracking, when an edge of the list of unprocessed edges has been processed (this is denoted by a call to `set_next_edge/0`). If the call to `same_branch/2` fails, the changes made by the current active edge are undone. The Prolog if-then-else (`-> ;`) is used to cut away unnecessary choice points. The `next_on_path/3` predicate unifies its last argument with the first edge that lies on the path between the edges given by its first two arguments.

To handle an active edge, we use the following rules, distinguishing between unprocessed and processed edges:

[2] If it is not already processed, the next edge's start node cannot exist.

```
active_edge(Edge) ==>
        edge_status(Edge,unprocessed) |
        edge_goal(Edge,Goal),
        call(Goal),
        (   true
        ;   activate_next_edge
        ).
active_edge(Edge) ==>
        edge_status(Edge,processed) |
        load_edge_changes(Edge),
        activate_next_edge.
```

The first rule handles an unprocessed edge. Its goal is called which causes a derivation that either ends in the creation of an end node (**Split** transition) in which case the next edge is activated automatically, or in a final execution state. If it ends in a successful final state, the next edge is activated on request (solution). We use the Prolog disjunction (`;`) for this purpose: the disjunctives are traversed left-to-right and depth first. If it ends in a failed final state, the derivation is backtracked automatically.

The second rule handles an already processed edge. For such an edge, the changes are loaded and it is again tried to activate the next unprocessed edge.

5.2 State Changes

An edge represents a derivation that connects its start node with its end node. For our purposes, it is sufficient to only look at edges that end in a choice point (i.e. an internal node). In [12], we show that every derivation can be written as $\langle A, S \cup S_-, B, T \rangle_n \rightarrowtail^*_{\omega_r} \langle A', S \cup S_+, B \wedge B_+, T \cup T_+ \rangle_{n+n_+}$. Here the sets S_+ and S_- contain respectively the added and removed CHR constraints. The conjunction B_+ contains the added built-in constraints, T_+ represents the new propagation history tuples and n_+ is the increase of the next free identifier number. Finally, the derivation changes the activation stack A into the new activation stack A'.

For the particular case of the derivation of an edge, we have $\langle [A_i \mid A], S \cup S_-, B, T \rangle_n \rightarrowtail^*_{\omega_r} \langle [A'_1 \vee \ldots \vee A'_m \mid A'], S \cup S_+, B \wedge B_+, T \cup T_+ \rangle_{n+n_+}$, where $[A_i \mid A]$ is one of the alternatives of the choice point represented by the edge's start node.

To be able to reconstruct this derivation, or in other words, to be able to move from one node to another, we need to be able to store how the CHR execution state is changed (i.e. create an explicit trail). We do not need the activation stack of the end node: the new unprocessed edges that start at this node (as created by a **Split** transition) already contain the information we need. Therefore, we only need to have a representation for the sets S_+, S_- and T_+, the conjunction B_+ and the integer n_+.

Our approach is similar to the explicit trailing mechanism of [16], but extends it by not only collecting changes to the CHR constraint store, but also to the built-in constraint store and the propagation history. We now describe how we can collect, store and load the changes.

Explicit Identifiers and Propagation History The identifiers of CHR constraints and the propagation history are not available to the CHR programmer. They are needed to create the sets S_-, S_+ and T_+, and to find the integer n_+. It is easy to add an explicit identifier to every constraint and to make the propagation history explicit. We show that this does not change the operational behavior of a CHR program in [12].

Changes to the CHR Constraint Store The CHR constraint store can change by adding identified constraints to it and by removing identified constraints from it. We show how these changes can be collected and stored and how we can load the stored changes. We use explicit identifiers for this purpose.

For every constraint `c/n` we create three new constraints: an *id-extended* constraint: `c_id/n+1`, a constraint to denote a constraint addition: `c_add/n+1` and a constraint to denote a constraint deletion: `c_del_n/1`. We add an explicit identifier and create a new element in S_+ by the following kind of rules:

```
c(X̄) <=>
        new_id(ID),
        c_add(ID,X̄),
        c_id(ID,X̄).
```

Here `new_id/1` creates a new unique identifier. Every simplification and simpagation rule of the form

```
r @ c1(X̄1), ..., ci(X̄i) \
    ci+1(X̄i+1), ..., cn(X̄n) <=>
        guard |
        body.
```

is changed into a rule

```
r @ normal, c_id1(ID1,X̄1), ..., c_idi(IDi,X̄i) \
    c_idi+1(IDi+1,X̄i+1), ..., c_idn(IDn,X̄n) <=>
        guard |
        c_del_ni+1(IDi+1),
        ...,
        c_del_nn(IDn),
        prelink(t(X̄1,...,X̄n)),
        body.
```

The `normal` constraint is used to turn rules on and off so that during the **Move Up** and **Move Down** transitions, no rules can fire, making these transitions behave as an atomic operation. For every removed constraint, an element of S_- is created. The use of the `prelink/1` predicate is explained further on, in the paragraph about the built-in constraint store.

Constraints that are added to and removed from the store on a single edge, must be removed from S_+. Therefore, for every constraint `c/n`, we create a rule:

```
c_del_n(ID), c_add(ID,_X1,...,_Xn) <=> true.
```

We can store the changes by collecting them in a list and saving the list in a non-backtrackable way.

Changes to the Propagation History Every propagation rule of the form

```
r @ c1(X̄1), ..., cn(X̄n) ==>
        guard |
        body.
```

is converted into a rule

```
r @ normal, c_id1(ID1,X̄1), ..., c_idn(IDn,X̄n) ==>
        \+ in_history(t(ID1, ..., IDn, r)),
        guard |
        add_to_history(t(ID1,...,IDn,r)),
        prelink(t(X̄1,...,X̄n)),
        body.
```

Because the propagation history is now directly available to us, it is easy to store how it is changed.

Changes to the Built-in Constraint Store For the built-in constraint store, we restrict ourselves to pure Prolog. In particular, we exclude features like attributed variables or global variables. Since the CHR implementation on which this work is based, is implemented using these features, we cannot support these features without losing the distinction between the CHR constraints and the built-in constraints.

In pure Prolog, the built-in store consists of a set of variable bindings, which can be represented as a list $[x_1 = t_1, \ldots, x_n = t_n]$ with x_i variables and t_i terms for $1 \leq i \leq n$. We need to be able to reconstruct the variable bindings for all relevant variables that appear in the derivation of the active edge. The relevant variables are the ones that are not *strictly local*, where strictly local variables are defined as the variables that do not occur in the initial goal, nor in any of the CHR constraints [1].

In the implementation, we construct two lists: one containing the variables $[x_1, \ldots, x_n]$ and one containing the terms $[t_1, \ldots, t_n]$. When a new variable appears, it is added to the list of variables. When finishing processing the active edge (because of a **Split** transition), this list of variables, which meanwhile has changed into a list of terms, is copied. On backtracking, the bindings of the variables are undone in the original list, but not in the copy.[3] This gives us the two lists that we need: the original list is the list of variables and the copy is the list of terms. The bindings can be reapplied by unifying the two lists.

The `prelink/1` predicate that we used in the transformation of the original program rules, adds the new variables in the term that is its argument, to the original list of variables. The transformation guarantees that all relevant variables are stored, because all variables that occur in the CHR constraint store and are bound, must be in the head of some rule.

[3] This is related to the way terms are constructed.

6 Evaluation

The implementation that has been presented in the previous section, has been used to transform some small example programs so that they are executed using depth first or breadth first search (i.e. both versions of the **Split** transition have been implemented).

6.1 Benchmarks

The following benchmark is performed on a 2.8 GHz Pentium IV processor using SWI-Prolog version 5.6.0. It forms a first indication of what is possible without having extra built-in support from the host language. A more fine-tuned implementation can probably still decrease the runtimes somewhat.

The benchmark consists of finding all solutions of a Sudoku puzzle with 295 different solutions. We measure the runtime, generated garbage and global stack (heap) space required for storing the search tree after the first solution is found. The search tree in this benchmark consists of 8 143 nodes and has a depth of 57.

Search Strategy	time(s)	garbage(MB)	global stack(MB)
Breadth First	50.14	420.82	2.22
Depth First (Explicit)	6.12	47.92	0.24
Depth First (Implicit)	2.04	1.08	0.12

Timings for some instances of the n queens problem can be found in [12]. The explicit version of depth first is using our program transformation, while the implicit version is using the built-in Prolog depth first search mechanism. The different timings between the explicit and implicit versions of depth first, show that there is a considerable overhead by making search explicit. This overhead is caused by the extra tasks of maintaining and storing an explicit trail and by making the propagation history and choice points explicit. The extra overhead of breadth first search is mostly related to loading the stored trails and in particular updating the internal hash tables. In this benchmark, these hash tables are responsible for at least a third of the runtime.

The amount of garbage generated is considerably larger in the explicit search strategies compared to implicit depth first. Moreover, it is much larger for breadth first than for depth first. This is explained by the fact that breadth first uses much more **Move Up** transitions: in the benchmark 113 391 times compared to 8 084 times for depth first.

Since all solutions of this benchmark are at the same depth, the part of the search tree that has to be in memory is maximal when reaching the first solution. This holds both for explicit depth first and explicit breadth first. Breadth first requires much more memory as all lower depth nodes that are potentially part of a full solution, have been visited and are in memory.

Finally, the benchmark clearly does not present breadth first search as a better alternative for depth first search. Although it is easy to find examples on which breadth first performs arbitrarily better than depth first, our aim is

showing what kind of overhead is generated by explicitly programming a different search strategy in CHR. The choice of good search strategies for various constraint programming problems is a completely different discussion.

6.2 Support from the Host Language

For different search strategies to become practical, they have to be implemented efficiently and it is clear that our source transformation is not highly performant. We can thus only conclude that extra support from the host language is necessary. In this subsection, we give our view of what is needed.

The main issue is to be able to jump from one node to another efficiently and to be able to return to previously visited nodes. We have implemented an explicit trailing mechanism for this, but such a mechanism can also be implemented at a lower level. An example of this is the XSB system [19] which supports tabling. Its SLG-WAM engine allows undoing changes on the trail without losing the information in the trail. Redoing changes is done by traversing the trail in a so-called forward mode. It also creates an explicit choice point stack. Instead of using an explicit trail, tabling can also be implemented using copying [5].

We think it is useful from a flexibility point of view, to have an explicit representation of choice points and low level primitives to move from one choice point to another. This supports more freedom to implement special search strategies. It might even be useful to allow the programmer to specify how the changes between different choice points should be stored: by copying, by trailing or not at all (recomputation). Finally, we note that host language support for the specification of different search strategies, can be easily extended to CHR implementations that are compiled into this host language, like CHR(Prolog).

7 Related Work

Adding different search strategies to declarative languages and in particular Logic Programming languages, has been done before. In [10], an operator for encapsulating search in the functional logic programming language Curry is presented. The relation with Prolog predicates like `findall/3` is given, noting that such predicates can return the solutions in any order if there are only finitely many, but cannot really search in a different than depth first order. The implementation of the search operator relies on a variable scoping mechanism, which is similar to having multiple copies of the variables. A similar idea is presented in Oz [18]. This language has a form of constraint store called a blackboard. The search operator creates local versions of this blackboard, with copies of the constraints that are on the global blackboard.

In [17] copying is combined with recomputation and compared with trailing. The Java Abstract Search Engine (JASE) which is part of the Java Constraint Kit (JACK) [2] expands on these ideas. It supports different search strategies amongst which breadth first search and restores states by using trailing, copying or recomputation, or by using a combination of these.

In Ciao Prolog [11], breadth first and iterative deepening search are supported using a source transformation. The breadth first implementation is based on the `findall/4` predicate,[4] which enumerates all alternatives, making copies of the variables. By using a limited form of interpretation, the alternative bodies of breadth first predicates are added to a list of alternatives, while for depth first predicates, all alternative solutions are added. This only works if the depth first predicate has only finitely many solutions.

Finally, in [4], the blackboard primitives of SICStus Prolog are used to implement intelligent backtracking in Prolog.

8 Conclusions and Future Work

We have formalized a theoretical semantics $\omega_t^\vee$ for CHR$^\vee$: CHR extended with disjunctions in rule bodies. We have introduced $\omega_r^\vee$, an extension of the refined operational semantics [6] of CHR towards CHR$^\vee$. This $\omega_r^\vee$ semantics supports the definition of different search strategies. These results extend the work in [3].

We refined our $\omega_r^\vee$ into an operationally equivalent formulation that is more suitable as a basis for trailing based implementations. We described how a breadth first search strategy can be implemented as a source to source transformation, using the currently available language features of the K.U.Leuven CHR system [15] and SWI-Prolog [20] only. This implementation introduces an explicit trailing mechanism that is an extension of the one in [16]. A first evaluation of the implementation has shown that the overhead created by the transformation is considerable and we have made suggestions to what kind of low level support from the host language could help to improve the performance.

Future Work Future work consists of adding low level support for search to the host language and the creation of a practical declarative framework for the specification of search strategies in CHR programs. We think this is essential if we want to use CHR to write highly performant CLP systems.

References

1. Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In Gert Smolka, editor, *CP'97: Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*, pages 252–266, Schloss Hagenberg, Austria, 1997. Springer Verlag.
2. Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauss. JACK: A Java constraint kit. In *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming, Kiel*, Kiel, Germany, September 2001.
3. Slim Abdennadher and Heribert Schütz. CHR$^\vee$: A flexible query language. In Troels Andreasen, Henning Christiansen, and Henrik Legind Larsen, editors, *FQAS*, volume 1495 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 1998.

[4] A difference list version of the more well-known `findall/3` predicate.

4. Maurice Bruynooghe. Enhancing a search algorithm to perform intelligent backtracking. *TPLP*, 4(3):371–380, 2004.
5. Bart Demoen and Konstantinos F. Sagonas. CAT: The copying approach to tabling. *Journal of Functional and Logic Programming*, 1999(Special Issue 2), 1999.
6. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In *ICLP'04: Proceedings of the 20th International Conference on Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104, St-Malo, France, Sep 2004. Springer Verlag.
7. Michael Fink, Hans Tompits, and Stefan Woltran, editors. *Proceedings of the 20th Workshop on Logic Programming*. INFSYS Research Report 1843-06-02 (TU Wien), 2006.
8. Thom W. Frühwirth. Constraint Handling Rules. In Andreas Podelski, editor, *Constraint Programming*, volume 910 of *Lecture Notes in Computer Science*, pages 90–107. Springer, 1994.
9. Michael Hanus. Adding Constraint Handling Rules to Curry. In Fink et al. [7], pages 81–90.
10. Michael Hanus and Frank Steiner. Controlling search in declarative programs. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *PLILP/ALP*, volume 1490 of *Lecture Notes in Computer Science*, pages 374–390. Springer, 1998.
11. Manuel V. Hermenegildo, Francisco Bueno, Daniel Cabeza, Manuel Carro, Maria J. García de la Banda, Pedro López-García, and Germán Puebla. The CIAO multi-dialect compiler and system: An experimentation workbench for future (C)LP systems. In Paqui Lucio, Maurizio Martelli, and Marisa Navarro, editors, *APPIA-GULP-PRODE*, pages 105–110, 1996.
12. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. Flexible search strategies for Prolog CHR. Technical Report CW 447, K.U.Leuven, Dept. CS, May 2006.
13. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. INCLP(R) - Interval-based nonlinear constraint logic programming over the reals. In Fink et al. [7], pages 91–100.
14. Ekkehard Krämer. A generic search engine for a Java constraint kit. Master's thesis, Institute of Computer Science, LMU, Munich, Germany, January 2001.
15. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In *First workshop on constraint handling rules: selected contributions*, pages 1–5, 2004.
16. Tom Schrijvers, Bart Demoen, Gregory Duck, Peter Stuckey, and Thom Frühwirth. Automatic implication checking for CHR constraints. In Horatiu Cirstea and Narciso Martí-Oliet, editors, *RULE'05: Proceedings of the 6th International Workshop on Rule-Based Programming*, Nara, Japan, April 2005.
17. Christian Schulte. Comparing trailing and copying for constraint programming. In *ICLP*, pages 275–289, 1999.
18. Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In Alan Borning, editor, *PPCP*, volume 874 of *Lecture Notes in Computer Science*, pages 134–150. Springer, 1994.
19. Terrance Swift and David Scott Warren. An abstract machine for SLG resulution: Definite programs. In *SLP*, pages 633–652, 1994.
20. Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, December 2003.

Extending CHR with Negation as Absence

Peter Van Weert, Jon Sneyers*, Tom Schrijvers**, Bart Demoen

Dept. of Computer Science, K.U.Leuven, Belgium
{petervw,jon,toms,bmd}@cs.kuleuven.be

Abstract. In this exploratory paper we introduce CHR$^{\neg}$, an extension of the CHR language with negation as absence, an established feature in production rule systems. Negation as absence is a procedural notion that allows a more concise and clean programming style. We propose a formal operational semantics for CHR$^{\neg}$ close to CHR's refined operational semantics. We illustrate and motivate its properties with examples.

1 Introduction

Constraint Handling Rules (CHR) [11, 17] is a high-level programming language extension based on multi-headed committed-choice rules. CHR is constraint-driven: *adding* constraints can cause rules to fire, depending on the *presence* of other constraints. The *removal* and *absence* of constraints has never received much attention in the past. This asymmetry stems from the original intended use of CHR: high-level and declarative prototyping of constraint solvers. In this context, constraints are not *removed* — they are merely *replaced* by equivalent and simpler constraints (hence the term "simplification rule").

However, CHR is increasingly used as a general programming language, in a wide range of applications [12, 18]. As a result, 'constraints' often correspond to elements in some data structure, operations that inspect or modify these elements, auxiliary programming constructs, flags, locks, loops, *etc.* Often, the absence of such 'constraints' is meaningful. Many CHR programmers are tempted to introduce auxiliary constraints and/or rules to test for the absence of constraints. Even worse, triggering rules on constraint removal requires numerous cross-cutting changes. Clearly, such ad-hoc solutions are very cumbersome and error-prone, and lead to more verbose and less declarative programs.

Closely related to CHR are *production rules* [9, 10, 15] (or *business rules*, as they are fashionably called now). These rule languages traditionally [8, 16] offer *negation as absence*. In this paper we introduce and explore an extension of CHR, called CHR$^{\neg}$, that allows negated heads in the left-hand side of rules. In the production rules literature, and in the rest of this paper, the words "negation", "negated", and "negatively" and the symbol "$\neg$" (pronounced *not*) are often used for negation *as absence*, even though negation as absence is not related to the classical logical negation (at least not in an obvious way).

* Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).
** Research Assistant of the Research Foundation - Flanders (F.W.O.-Vlaanderen).

Overview. In Section 2 we start by introducing CHR$^\neg$ using a series of clarifying examples. Next, Section 3 defines the formal operational semantics of CHR$^\neg$, followed by a critical discussion in Section 4. Finally, Section 5 concludes.
An extended version of this paper is available as a technical report [20].

2 CHR$^\neg$ by Example

In this section we introduce CHR$^\neg$ by means of several small examples. The examples are valid under the refined operational semantics $\omega_r^\neg$ of CHR$^\neg$, which we describe informally in this section and formally in the next one. For now, it suffices to know that it is based on the conventional operational semantics ω_r of CHR [6]. First we extend the regular CHR syntax [11, 13] with negated heads.

2.1 Syntax and terminology

The general form of a CHR$^\neg$ rule is:

$$name \ @ \ H_1 \setminus H_2 \setminus\!\setminus N_1 \mid G_1 \setminus\!\setminus \ \ldots \ \setminus\!\setminus N_k \mid G_k \Longleftrightarrow G \mid B \qquad (1)$$

The *heads* (H_1, H_2, N_1, …, N_k) are conjunctions of CHR constraints. The heads inherited from regular CHR — H_1 (the *kept* constraints) and H_2 (the *removed* constraints) — are referred to as *positive heads*. Depending on the context, we also use the term *positive head* to denote $H_1 \wedge H_2$. The N_i (for $i \in \{1, \ldots, k\}$) are *negated heads*. If $k = 0$ the rule has no negated heads. Each negated head N_i has a *negated guard* G_i. Negated heads are not allowed to be empty. Like in regular CHR, if one of H_1 (in case of a *simplification rule*) or H_2 (*propagation rule*) is empty, we omit the " $\setminus$ ". For propagation rules we use "$\Longrightarrow$" instead of "$\Longleftrightarrow$". At least one of the positive heads has to be non-empty. The *guards* (the *positive guard* G and the negated guards G_i) are conjunctions of built-in constraints. Empty guards (*true*) may be omitted together with the " | " symbol. Finally, the *body* (B) is a conjunction of built-in and CHR constraints.

2.2 Negated heads as extra precondition

We now present the preconditions for applying a CHR$^\neg$ rule. When and whether an applicable rule will actually be applied is determined by the operational semantics (covered in Sections 2.3 and 3).

A CHR$^\neg$ rule without negated heads is applicable if it would be in regular CHR. A negated head adds an extra precondition: a rule may *not* be applied if the constraint store contains CHR constraints that match the negated head. More precisely: a rule of the form (1) is applicable if

$$\exists \bar{p} \ \exists S_r \Big(S = H_1 \uplus H_2 \uplus S_r \ \ \wedge \ \ G \ \ \wedge \ \ \textstyle\bigwedge_{i=1}^{k} \neg \exists \bar{n}_i \big(N_i \subseteq\!\!\!\!\cdot \ S_r \ \wedge \ G_i \big) \Big)$$

where S is a multiset representing the constraint store, $\bar{p}$ are the variables occurring in H_1, H_2, or G, and $\bar{n}_i$ are the variables occurring in N_i or G_i but not in $\bar{p}$ (the *positive* variables), and $\uplus$ and $\subseteq\!\!\!\!\cdot$ denote multiset union and subset.

Example 1. The following CHR$^\neg$ rule expresses that "*If* a person `X` is not married, *then* that person is single": `person(X) \\ married(X) ==> single(X).`

Variables bound by the positive head or guard (e.g. `X` in the above example) can be used in a negated head. Using variables introduced in the positive *guard* (this arguably is rarely needed) breaks the left-to-right reading of a CHR$^\neg$ rule. By contrast, a negated head *cannot* bind variables to be used in the right-hand side of a rule. The intuition is that a negated head describes something that *is not there*. Therefore the scope of a variable defined in a negated head is limited to the head itself (and its guard):

Example 2 (Variable scope). `find_singles \\ married(X) ==> single(X).`

The rule in this example is applicable if `find_singles` is in the store and there is *no* `married/1` constraint in the store *at all*. When applied, it adds a `single(X)` constraint, where `X` is a fresh variable. This is probably not the intended meaning.

Example 3. Negated heads allow more concise and declarative programs. Consider the following rules from an implementation of Dijkstra's algorithm [18]:

```
dist(N,L), edge(N,N2,W) ==> L2 is L+W, relabel(N2,L2).
dist(N,_) \ relabel(N,_) <=> true.
relabel(N,L) <=> doi(N,L).
```

These rules can be rewritten to just one rule, eliminating both the auxiliary constraint `relabel/2` and the dependency on the execution order of ω_r:

```
dist(N,L), edge(N,N2,W) \\ dist(N2,_) ==> L2 is L+W, doi(N2,L2).
```

Example 4 (Distinct constraints). Consider the following CHR$^\neg$ program:

```
parent(X,Y) \ parent(X,Y) <=> true.
parent(X,Y) \\ parent(X,_) ==> only_child(Y).
```

The first rule states that we never keep more than one constraint to describe that "`X` is a parent of `Y`". We say the `parent` constraint has *set semantics* (in contrast to the default *multiset* semantics). For the simpagation rule to be applicable, CHR requires the two constraint instances matching the positive head to be different (with identical arguments). For negated heads, we opted for something similar: constraints already used in the matching of the positive head are not allowed in the matching of the negated head. Hence, the second rule expresses "*If* `Y` is a child of `X` and there is no *other* child of `X`, *then* `Y` is an only child". We refer to this as the *distinct constraints* matching strategy (see [20]).

Example 5. The rule `p \\ p ==> q` should not be read as "*If* `p` and ¬`p`, *then* `q`" (a rule for which the antecedent never holds), since we are introducing negation *as absence* and not the classical logical negation. It should also not be read as "*If* `p` is present and `p` is absent, *then* add `q`" (a rule for which the antecedent also never holds), because of the *distinct constraints* matching strategy. A correct reading is: "*If* `p` is present, and there is no *other* `p` in the store, *then* add `q`", or, in other words: "*If* there is *exactly one* `p` constraint, *then* add `q`".

Negated guards. Up till now we have only considered negated heads without guards — although non-singleton variables in (negated) heads are implicit guards that can be made explicit: for example, the rule in Example 1 is short for:

```
person(X) \\ married(X0) | X0 == X  ==> single(X).
```

Example 6. In general, a negated head can have an arbitrary guard. The following CHR$^\neg$ rule defines the query-constraint `get_min/1`:

```
c(X) \ get_min(Min) \\ c(Y) | Y < X <=> Min = X.
```

This rule reads: "*If* collection `c` contains an element `X` and there is no (other) element `Y` in `c` with `Y < X`, *then* `X` is the minimum". Without negation:

```
c(X) \ get_min(Min) <=> current(X,Min).
c(X) \ current(Current,Min) <=> X < Current | current(X,Min).
current(Current,Min) <=> Min = Current.
```

Negated conjunctions. Like a positive head, a negated head is a *conjunction.*

Example 7. A half-sibling is a sibling by one parent but not by both:

```
parent(P1,X), parent(P1,Y) \\ parent(P2,X), parent(P2,Y)
 ==> half_sibling(X, Y).
```

Multiple negated heads. So far we have only considered CHR$^\neg$ rules with a single negated head. In general, any number of negated heads can be used.

Example 8. If a parent of `X` is married to a parent of `Y`, but `X` and `Y` do not share a (biological) parent, then `X` and `Y` are step-siblings:

```
parent(P1,X), parent(P2,Y), married(P1,P2)
   \\ parent(P2,X)   \\ parent(P1,Y)     ==> step_sibling(X,Y).
```

Example 9 (Variable scope 2). The new variables in a negated head are local:

```
c(L), c(U) \ get_bounds(LB,UB)
   \\ c(Z) | Z < L   \\ c(Z) | Z > U     <=> LB = L, UB = U.
```

The last line in the above rule is equivalent to:

```
   \\ c(Z1) | Z1 < L   \\ c(Z2) | Z2 > U   <=> LB = L, UB = U.
```

2.3 Triggering on negated heads

We know now when CHR$^\neg$ rules with negated heads are *applicable.* In this section we aim to give some intuition about when they should be *applied* (in anticipation of Section 3, where we formally define the operational semantics of CHR$^\neg$).

Essentially, the operational semantics of CHR [6] determines that a rule can fire whenever a CHR constraint is added that occurs in its (positive) head, or when a change in the built-in constraint store causes its (positive) guard to succeed. In the latter case we say that the guard *triggers* the rule. We could simply keep this semantics and treat negated heads as an extra, *passive* condition of rule applicability, much like a positive guard that does not trigger. But then many applicable CHR$^\neg$ rules would never be applied.

We allow a rule to fire, not only when its positive head and guard become satisfied, but also when its guarded negated head(s) become satisfied. Symmetric to the positive case, we distinguish two causes for a guarded negated head to become satisfied. The first, and most obvious, is the removal of one of the negatively occurring constraints. The second is related to negated guards. We say that the removal resp. the negated guard *triggers* the rule.

Triggering on removal. Figure 1 lists a CHR$^\neg$ program to maintain reachability information in a directed graph. The constraint store contains `node/1` and `edge/2` constraints, representing a dynamic graph. It is useful to imagine the listed program as part of a larger program that inserts and removes `node/1` and `edge/2` constraints at arbitrary points throughout the program.

The `set_sem_*` rules enforce set semantics. Only the `edge/2` constraint has a multiset semantics, implying that parallel edges are allowed. We use an auxiliary constraint `path(From,Over,To)` to represent that a path exists between node `From` and node `To` that starts with `edge(From,Over)`. The rules `trivial_path` and `extend_path` generate `path(A,_,B)` constraints for every path between `A` and `B`. The complementary rules `broken_tr` and `broken` ensure that, if a path is broken (because some edge or node is removed), all `path` constraints depending on the broken path are removed recursively. Finally, the last three rules use a

reachability.chr

```
set_sem_nodes @ node(A) \ node(A) <=> true.
missing_from  @ edge(A,_) \\ node(A) <=> true.
missing_to    @ edge(_,B) \\ node(B) <=> true.

set_sem_paths @ path(A,B,C) \ path(A,B,C) <=> true.
trivial_path @ node(A) ==> path(A,A,A).
extend_path  @ edge(A,B), path(B,_,C) ==> A \== B, A \== C | path(A,B,C).
broken_tr    @ path(A,_,_) \\ node(A) <=> true.
broken       @ path(A,B,C) \\ edge(A,B), path(B,_,C) <=> A \== B | true.

set_sem_reaches @ reaches(A,B) \ reaches(A,B) <=> true.
path            @ path(A,_,B) ==> reaches(A,B).
no_paths        @ reaches(A,B) \\ path(A,_,B) <=> true.
```

Fig. 1. CHR$^\neg$ program to maintain reachability in dynamic directed graphs

similar pattern to make sure that there is a `reaches(A,B)` constraint in the store if *and only if* a path exists between `A` and `B`.

This program illustrates the usefulness of triggering on removal: not only can we maintain correct reachability information if new paths are *created* (by adding edges or unifying nodes), in CHR$^\neg$ we can write *complementary rules* to keep this information *consistent* if edges (and thus paths) are *removed*. Another usage pattern shown is what we could call *garbage collection*. Obviously there is no need to keep an edge constraint if one of its end nodes is removed. In CHR$^\neg$ we can easily add rules (`missing_from` and `missing_to`) to remove such edges. This removal then recursively triggers the removal of all other redundant information (paths, reachability, ...).

Example 10. Consider adding the following rule to the program in Figure 1:

```
strongly_connected, node(A), node(B) \\ reaches(A,B) ==> edge(A,B).
```

This rule forces the graph to be strongly connected if the `strongly_connected` constraint is in the store. It adds edges until every node reaches every other node. When edges (or nodes) are removed, the resulting removal of `reaches/2` constraints triggers the addition of edges until strong connectivity is restored.

Example 11. A well-known CHR pattern to maintain the minimal element of a collection of `c/1` constraints consists of the following two rules:

```
c(X) ==> min(X).
min(X) \ min(Y) <=> X ≤ Y | true.
```

Unfortunately these rules do not consistently keep the minimum if elements are removed. In CHR$^\neg$ we can generalize the above pattern to deal with removal:

```
min(X) \\ c(X) <=> true.
c(X) \\ c(Y) | Y < X ==> min(X).
min(X) \ min(Y) <=> X ≤ Y | true.
```

When the minimal element `c(X)` is removed, the corresponding `min(X)` constraint is removed and replaced by a new minimum.

Triggering on negated guards. A guarded negated head can also become satisfied by a (non-monotonic) change in the built-in store. The addition of a built-in constraint may cause the guard of a negated head to become false for all negated head matchings.

Example 12. Consider this slightly altered version of the last rule of Example 11

```
c(X) \\ c(Y) | Y @< X ==> min(X).
```

where `'@<'` is the (Prolog) built-in for *less than in the standard order of terms*. Consider the query "`c(A), c(B), A=g`". Adding `c(A)` propagates `min(A)`. Say `A @< B`, so adding `c(B)` does not affect the minimum. Then, unifying `A` with `g` grounds the first constraint. This should trigger the above rule, since now the negated head is satisfied for `c(B)` — not because `c(g)` is removed, but because `g @< B` no longer holds (while `A @< B` did).

This type of triggering can only happen if guards behave non-monotonically, i.e. are true and become false. In host-languages like Prolog, such guards are rather exceptional (which is why the above example is a little far-fetched). Other examples of non-monotonic guards in Prolog are "`var(X)`" which becomes false when `X` is instantiated, "`X \== Y`" which becomes false when `X` and `Y` are unified, and dynamic predicate calls in programs that use `retract/1`. In other host languages (e.g. Java [21]), this type of guards might occur more frequently.

3 Formal Semantics

In this section we formalize the semantics of a CHR$^\neg$ program by defining a refined operational semantics. This semantics reflects several decisions already motivated in the previous section, like rule applicability and triggering of negated heads. Other aspects covered in this section are execution order and the role of the propagation history. These aspects are illustrated in an example derivation.

The formulation of the semantics is based on [6], where Duck et al. presented a similar refined semantics ω_r for regular CHR. Much like [6], we also defined a theoretical operational semantics for CHR$^\neg$, and proved that the refined semantics presented in this section is an instance thereof. These results can be found in the extended version of this paper [20].

3.1 $\omega_r^\neg$: the refined operational semantics of CHR$^\neg$

The $\omega_r^\neg$ semantics is formulated as a state transition system. Transition rules define the relation between subsequent execution states in a CHR$^\neg$ derivation. We use $+\!\!+$ for sequence *concatenation* and $\sqcup$ for *disjoint union* of sets.

Execution state. Formally, an execution state of $\omega_r^\neg$ is a tuple $\sigma = \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$. All parts besides the *execution stack* $\mathbb{A}$ are defined as in [6]. The CHR constraint *store* $\mathbb{S}$ is a set of *identified* CHR constraints that can be matched with the rules. An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with a unique *constraint identifier* i which is used to distinguish otherwise identical constraints. The counter n represents the next free integer that can be used to identify a CHR constraint. We introduce the functions $chr(c\#i) = c$ and $id(c\#i) = i$, and extend them to sequences and sets in the obvious manner. The identifiers are used by the *propagation history* $\mathbb{T}$, a set of tuples, each recording the identities of the CHR constraints that fired a rule, and the name of the rule itself. The primary function of this history is to prevent trivial non-termination for propagation rules; its role in $\omega_r^\neg$ will be further analyzed later in this section. Finally, the *built-in constraint store* $\mathbb{B}$ is an abstract logical conjunction of constraints, modeling all built-in constraints that have been passed to the underlying solver.

The execution stack $\mathbb{A}$ is used to treat constraints as procedure calls — as in ω_r. The top-most element is called the *active constraint*. Each newly added CHR constraint initiates a search for partner constraints to match the heads of the

rules. Adding a built-in constraint initiates a similar search for applicable rules. As with a procedure, when a rule fires, other constraints (its body) are executed, and execution does not return to the current active constraint *until* these calls have finished. This approach is used because it corresponds closely to that of the stack-based programming languages to which CHR is compiled. Like ω_r, the $\omega_r^{\neg}$ semantics fixes the order in which searches for matching rules are conducted. That is why constraints on the execution stack can become *occurrenced*. The *positive* occurrences of constraints in the heads of the rules are numbered in a top-down, right-to-left manner . An *occurrenced* identified CHR constraint $c\#i:j$ indicates that only matches with positive occurrence j of constraint c should be considered when the constraint is active.

Thus far, the execution state corresponds to that of ω_r. However, in CHR$^{\neg}$, rules also trigger on constraint *removal*. This translates to a slight modification in the definition of the execution stack: formally, $\mathbb{A}$ is a sequence of constraints, (occurrenced) identified CHR constraints *and (occurrenced) negated CHR constraints. Negated CHR constraints* are denoted as $\neg c$, where c is a CHR constraint (this notation is extended to sequences in the usual way). An *occurrenced* negated CHR constraint $\neg c:j$ denotes that *the j-th rule where c occurs negatively* is considered.

Example 13. Recall from Example 11 the CHR$^{\neg}$ pattern to maintain the minimum of a collection. These three CHR$^{\neg}$ rules rely on rule order to renew the minimum after the current minimal element is removed: *first* the old minimum is removed by the first rule, *then* the second rule adds the new minimum. Both rules trigger on the same removal. If the second rule would be applied first, we could get undesired behavior: two `min/1` constraints would be in the store, which allows application of the third rule, removing the *new* minimum!

Transition rules. Given an initial goal sequence G, the *initial execution state* σ_0 is $\langle G, \emptyset, true, \emptyset\rangle_1$. Execution proceeds by exhaustively applying transitions to σ_0, until the built-in store is unsatisfiable or no more transitions are applicable.

The transition rules of $\omega_r^{\neg}$ are listed in Figure 2. Besides the relatively straightforward extensions to deal with negation, the most important addition to ω_r is the **AllowReapply** transition. We say a (propagation!) rule *reapplies* if it applies with a combination of positive constraints that has already caused the rule to fire earlier in the derivation. In ω_r, rules never reapply because of the propagation history. In $\omega_r^{\neg}$ however, a rule can become reapplicable because of the **AllowReapply** transition. This new transition removes a propagation history tuple as soon as the rule, whose application introduced the tuple, is no longer applicable. Because the rule has to be inapplicable, trivial nontermination caused by infinite propagation is still avoided — the original motivation for the history.

There are two ways to get a situation in which **AllowReapply** is applicable: 1) one of the constraints used in the matching of the positive head has been removed from the constraint store; or 2) the applicability condition no longer holds for the particular rule and combination of constraints given in the tuple. Clearly the first case is a kind of garbage collection and does not affect derivations (cf.

0. AllowReapply $\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T}' \rangle_n$ where $\mathbb{T}' = \mathbb{T} \setminus \{h\}$ and $\forall \mathbb{A}'' : \langle \mathbb{A}'', \mathbb{S}, \mathbb{B}, \mathbb{T}' \rangle_n \not\rightarrowtail \langle \mathbb{A}', \mathbb{S}', \mathbb{B}', \mathbb{T} \rangle_n$. No other transition is applicable if this one is.

---

1. Solve $\langle [b|\mathbb{A}], S_0 \sqcup S_1, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail \langle \{[x, \neg x] | x \in S_1\} +\!\!+ \mathbb{A}, S_0 \sqcup S_1, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$ where b is a built-in constraint and $vars(S_0) \subseteq fixed(\mathbb{B})$, the variables fixed by $\mathbb{B}$.

---

2. Activate Positive $\langle [c|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail \langle [c\#n\!:\!1|\mathbb{A}], \{c\#n\} \sqcup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$ where c is a CHR constraint.

---

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

---

4. Activate Negated $\langle [\neg c|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail \langle [\neg c\!:\!1|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$

---

5. Drop $\langle [x : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ where x is a (positive) identified CHR constraint $c\#i$ and there is no j-th positive occurrence of c, or x is a negated constraint $\neg c$ and there are less than j rules where c occurs negatively.

---

6. Simplify $\langle [c\#i\!:\!j|\mathbb{A}], \{c\#i\} \sqcup H_1 \sqcup H_2 \sqcup H_3 \sqcup S, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail$
$\langle \neg(H_2' +\!\!+ [d] +\!\!+ H_3') +\!\!+ B +\!\!+ \mathbb{A}, H_1 \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T}' \rangle_n$ where the j-th (positive) occurrence of c is d in a (renamed apart) rule r of the form

$$r \ @ \ H_1' \setminus H_2', d, H_3' \setminus\!\!\setminus N_1' \mid G_1 \setminus\!\!\setminus \ \ldots \ \setminus\!\!\setminus N_k' \mid G_k \Longleftrightarrow G \mid B$$

and there exists a matching substitution θ such that $c = \theta(d)$, $chr(H_i) = \theta(H_i')$, and $applicable(G, \bar{N}', \bar{G}, \mathbb{B}, \theta, S)$ holds.
Let $t = (id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3), r)$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

---

7. Propagate $\langle [c\#i\!:\!j|\mathbb{A}], \{c\#i\} \sqcup H_1 \sqcup H_2 \sqcup H_3 \sqcup S, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail$
$\langle \neg(H_3') +\!\!+ B +\!\!+ [c\#i\!:\!j|\mathbb{A}], \{c\#i\} \sqcup H_1 \sqcup H_2 \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T}' \rangle_n$ where the j-th (positive) occurrence of c is d in a (renamed apart) rule r of the form

$$r \ @ \ H_1', d, H_2' \setminus H_3' \setminus\!\!\setminus N_1' \mid G_1 \setminus\!\!\setminus \ \ldots \ \setminus\!\!\setminus N_k' \mid G_k \Longleftrightarrow G \mid B$$

and there exists a matching substitution θ such that $c = \theta(d)$, $chr(H_i) = \theta(H_i')$, and $applicable(G, \bar{N}', \bar{G}, \mathbb{B}, \theta, S)$ holds.
Let $t = (id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3), r)$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

---

8. TriggerNegated $\langle [\neg c\!:\!j|\mathbb{A}], H_1 \sqcup H_2 \sqcup S, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail$
$\langle \neg(H_2') +\!\!+ B +\!\!+ [\neg c\!:\!j|\mathbb{A}], H_1 \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T}' \rangle_n$ where

$$r \ @ \ H_1' \setminus H_2' \setminus\!\!\setminus N_1' \mid G_1 \setminus\!\!\setminus \ \ldots \ \setminus\!\!\setminus N_k' \mid G_k \Longleftrightarrow G \mid B$$

is a renamed apart instance of the j-th rule where c occurs negatively, and there exists a matching substitution θ such that $chr(H_i) = \theta(H_i')$, and $applicable(G, \bar{N}', \bar{G}, \mathbb{B}, \theta, S)$ holds. Let $t = (id(H_1) +\!\!+ id(H_2), r)$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

---

9. Default $\langle [x\!:\!j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail \langle [x\!:\!j+1|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$ if the current state cannot fire any other transition (x can be either positive ($c\#i$) or negated ($\neg c$)).

The *applicability condition* is defined as follows: $applicable(G, \bar{N}, \bar{G}, \mathbb{B}, \theta, S) =$

$$\mathcal{D}_b \models \mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}} \Big(\theta(G) \wedge \textstyle\bigwedge_{i=1}^{k} \Big(\forall X \subseteq S : \neg \exists \eta : chr(X) = (\eta\theta)(N_i) \wedge (\eta\theta)(G_i) \Big) \Big)$$

where $\mathcal{D}_b$ is the built-in constraint domain and η are matching substitutions.

Fig. 2. The transition rules of the refined operational semantics $\omega_r^{\neg}$ for CHR$^{\neg}$.

CleanHistory in [19]). However, in the second case, removing history tuples may allow reapplication of a rule. Consider a rule of the form $P \setminus\setminus N \Longrightarrow B$. Suppose this rule is applied at some point in the derivation. Later on, the constraint N is added, so the rule becomes inapplicable and the **AllowReapply** transition is applied. Now, if N is removed, the rule can be applied *again* (provided, of course, that P is still in the store). Another example of reapplication is given in Section 3.2.

There are also less obvious ways for a rule to become reapplicable: 1) a non-monotonic negated guard: a guard which is first false, allowing rule application, then becomes true, resulting in an **AllowReapply**, and then becomes false again, causing rule reapplication; or 2) a non-monotonic positive guard — a guard which is true, then false, then true again. The second case is why $\omega_r^\neg$ is *not* a generalization of ω_r (for programs with non-monotonic guards): the **AllowReapply** transition could allow the reapplication of propagation rules *without negated heads*. However, the semantics without **AllowReapply** *is* a generalization of ω_r. We call this the *fire-once* refined operational semantics $\omega_r^{\neg 1}$.

Determinism. Like ω_r, the $\omega_r^\neg$ semantics only has two sources of non-determinism: the order in which the constraints are added to $\mathbb{A}$ in the **Solve** transition and the choice of partner constraints in **Propagate**, **Simplify** and **TriggerNegated**. The **AllowReapply** transition does not introduce additional non-determinism, because we require it to fire *immediately* when it becomes applicable.

3.2 Example derivation

Consider once more the CHR$^\neg$ program of Example 11. Suppose we name these rules $\mathtt{r}_1$ to $\mathtt{r}_3$ and add the following rule to remove an element:

```
remove @ rm(X), c(X) <=> true.
```

Figure 3 shows the derivation for the query `c(9), c(5), rm(5)`. The derivation starts by activating `c(9)`, which fires the second rule $\mathtt{r}_2$. This adds `min(9)`, which does not fire any rule. Note that firing $\mathtt{r}_2$ has added a tuple to the propagation history. After `c(9)` is dropped, the next constraint `c(5)` gets activated. Because this inserts `c(5)` into the store, rule $\mathtt{r}_2$ is no longer applicable for `c(9)`, so **AllowReapply** removes the corresponding history tuple. Next, $\mathtt{r}_2$ fires once more and adds `min(5)`. This causes r_3 to remove the old minimum, `min(9)`#2, from the store, also adding ¬`min` on top of the stack (which does not trigger any rule since `min/1` does not occur negatively). Because negated constraints are not matched, the constraint symbol alone suffices. An **AllowReapply** transition removes the newly added history tuple — this always happens after application of non-propagation rules since they immediately become inapplicable (because they remove (some of) their positive heads).

We now jump to the first time constraint removal triggers a rule. After `c(5)` is removed, `min(5)` is removed by rule r_1. This removal itself triggers nothing, but the removed `c(5)` constraint also triggers the next rule, r_2, which adds the new

$\langle [\mathtt{c(9)}, \mathtt{c(5)}, \mathtt{rm(5)}]\ ,\ \emptyset\ ,\ \emptyset\rangle_1$

$\rightarrowtail_{act}\ \langle [\mathtt{c(9)\#1:1}, \mathtt{c(5)}, \mathtt{rm(5)}]\ ,\ \{\mathtt{c(9)\#1}\}\ ,\ \emptyset\rangle_2$

$\rightarrowtail_{prop}\ \langle [\mathtt{min(9)}, \mathtt{c(9)\#1:1}, \ldots]\ ,\ \{\mathtt{c(9)\#1}\}\ ,\ \{([1], \mathtt{r_2})\}\rangle_2$

$\rightarrowtail_{act}\ \langle [\mathtt{min(9)\#2:1}, \ldots]\ ,\ \{\mathtt{c(9)\#1}, \mathtt{min(9)\#2}\}\ ,\ \{([1], \mathtt{r_2})\}\rangle_3$

$\rightarrowtail^{*}\ \langle [\mathtt{c(9)\#1:1}, \mathtt{c(5)}, \mathtt{rm(5)}]\ ,\ \{\mathtt{c(9)\#1}, \mathtt{min(9)\#2}\}\ ,\ \{([1], \mathtt{r_2})\}\rangle_3$

$\rightarrowtail^{*}\ \langle [\mathtt{c(5)\#3:1}, \mathtt{rm(5)}]\ ,\ \{\mathtt{c(9)\#1}, \mathtt{min(9)\#2}, \mathtt{c(5)\#3}\}\ ,\ \{([1], \mathtt{r_2})\}\rangle_4$

$\rightarrowtail_{ara}\ \langle [\mathtt{c(5)\#3:1}, \mathtt{rm(5)}]\ ,\ \{\mathtt{c(9)\#1}, \mathtt{min(9)\#2}, \mathtt{c(5)\#3}\}\ ,\ \emptyset\rangle_4$ $\hookleftarrow$

$\rightarrowtail_{prop}\ \langle [\mathtt{min(5)}, \mathtt{c(5)\#3:1}, \mathtt{rm(5)}]\ ,\ \{\mathtt{c(9)\#1}, \mathtt{min(9)\#2}, \mathtt{c(5)\#3}\}\ ,\ \{([3], \mathtt{r_2})\}\rangle_4$

$\rightarrowtail^{*}\ \langle [\mathtt{min(5)\#4:3}, \ldots]\ ,\ \{\mathtt{c(9)\#1}, \mathtt{min(9)\#2}, \mathtt{c(5)\#3}, \mathtt{min(5)\#4}\}\ ,\ \{([3], \mathtt{r_2})\}\rangle_5$

$\rightarrowtail_{prop}\ \langle [\neg\mathtt{min}, \mathtt{min(5)\#4:3}, \ldots]\ ,\ \{\mathtt{c(9)\#1}, \mathtt{c(5)\#3}, \mathtt{min(5)\#4}\}\ ,\ \{([3], \mathtt{r_2}), ([4,2], \mathtt{r_3})\}\rangle_5$

$\rightarrowtail_{ara}\ \langle [\neg\mathtt{min}, \mathtt{min(5)\#4:3}, \ldots]\ ,\ \{\mathtt{c(9)\#1}, \mathtt{c(5)\#3}, \mathtt{min(5)\#4}\}\ ,\ \{([3], \mathtt{r_2})\}\rangle_5$

(we now jump to a state little after `c(5)` is removed by `rm(5)`)

$\rightarrowtail^{*}\ \langle [\neg\mathtt{c:1}, \mathtt{rm(5)\#5:1}]\ ,\ \{\mathtt{c(9)\#1}, \mathtt{min(5)\#4}\}\ ,\ \emptyset\rangle_6$

$\rightarrowtail_{trneg}\ \langle [\neg\mathtt{min}, \neg\mathtt{c:1}, \ldots]\ ,\ \{\mathtt{c(9)\#1}\}\ ,\ \emptyset\rangle_6$

$\rightarrowtail^{*}\ \langle [\neg\mathtt{c:2}, \ldots]\ ,\ \{\mathtt{c(9)\#1}\}\ ,\ \emptyset\rangle_6$

$\rightarrowtail_{trneg}\ \langle [\mathtt{min(9)}, \neg\mathtt{c:2}, \ldots]\ ,\ \{\mathtt{c(9)\#1}\}\ ,\ \{([1], \mathtt{r_2})\}\rangle_6$ $\hookleftarrow$

$\rightarrowtail^{*}\ \langle []\ ,\ \{\mathtt{c(9)\#1}, \mathtt{min(9)\#6}\}\ ,\ \{([1], \mathtt{r_2})\}\rangle_7$

Fig. 3. $\omega_r^{\neg}$ derivation for "`c(9), c(5), rm(5)`" (the built-in store $\mathbb{B}$ is not included).

minimum. Recall from Example 13 how we rely on *rule order* here. Finally, the new minimum is activated and added to store, and the derivation reaches a final state. One final remark: the reason r_2 could fire *again* with $\mathtt{c(9)\#1}$ matching its positive head, is because the **AllowReapply** rule removed the corresponding history tuple earlier in the derivation (we marked the relevant transitions with "$\hookleftarrow$"). This illustrates why the **AllowReapply** transition is necessary.

3.3 Implementation

We have designed and implemented a source-to-source transformation scheme from CHR$^{\neg}$ to regular CHR. In general, a CHR$^{\neg}$ program with r rules and c constraints requires $\mathcal{O}(rc)$ CHR rules using this scheme. The full transformation is described in detail in [20]. This prototype implementation has enabled us to experiment with CHR$^{\neg}$ and to gain insight in its subtleties. In particular, all examples of Section 2 are executable in our prototype implementation of CHR$^{\neg}$.

4 Discussion

In this section we discuss some theoretical and practical disadvantages and limitations of the $\omega_r^{\neg}$ semantics. We also propose approaches to overcome them.

Lost logical reading. Besides the *operational* semantics ω_t [11] and ω_r [6], CHR without negation features a *declarative* semantics: CHR rules have a *logical reading*, i.e. they correspond (in a straightforward way) to formulae in either

classical first-order predicate logic [11], or linear logic [4]. This useful property is lost for CHR$^{\neg}$ since negation as absence is an inherently operational concept: it seems to be impossible to integrate it in any (monotonic) logic in a natural way (much like negation as failure in logic programming languages).

Lost monotonicity. Another fundamental property of the CHR language is its monotonicity [2]: rule application is independent of context, so adding constraints cannot inhibit the applicability of a rule. The monotonicity property can be stated as follows: "if $A \rightsquigarrow B$, then $A \wedge C \rightsquigarrow B \wedge C$ for any C". Clearly, this property is lost in CHR$^{\neg}$ — and with it all results that rely on it, most notably the results on confluence [2] and the notions of parallelism introduced in [12].

Unexpected behavior. Programming in CHR$^{\neg}$ turns out to be challenging: the $\omega_r^{\neg}$ semantics often results in unexpected and somewhat counterintuitive program behavior. Some examples:

Example 14 (Destructive update). Let `account(X,B)` model some novel type of bank account of a `client(X)` with balance `B`. Suppose the bank wants to send an information brochure to all (new) clients that do not yet have this type of account. They add the following CHR$^{\neg}$ rule:

```
send @ client(X) \\ account(X,_) ==> send_brochure_to(X).
```

Suppose the CHR$^{\neg}$ program used by the bank also contains a rule of the form "`deposit(X,Amount), account(X,B) <=> account(X,B+Amount)`". This rule is an example of a common CHR pattern, often referred to as *(destructive) update*. The problem is that removing a constraint (`account(X,B)`) can trigger rules (`send`), even though the constraint will immediately be replaced with an updated version. This means that each time a client deposits an amount on his account, he receives a brochure. A brochure about an account type he already has. Some might argue this is not what the bank intended.

Example 15 (Order). Negatively occurring constraints have to be added in the right order. The following program causes each child to be declared orphan at birth because of an incorrect order in the body of the first rule:

```
birth(C,F,M) <=> child(C), father(F,C), mother(M,C).
child(C) \\ father(_,C) \\ mother(_,C) ==> orphan(C).
```

Rearranging the body constraints solves this problem. Another example is the graph program (Figure 1), where nodes have to be created before edges. In some programs there is no correct order to add constraints one at a time: in Example 4, there is no way to insert constraints `parent(P,`C_1`)` and `parent(P,`C_2`)` without propagating `only_child(`C_1`)` or `only_child(`C_2`)`. We expect that this can often be corrected: e.g. in Example 4, we could add the rule:

```
parent(P,C), parent(P,_) \ only_child(C) <=> true.
```

Example 16 (Atomicity). Suppose we want to improve the graph program of Figure 1 by adding (in front of the program):

```
add_nodes @ edge(A,B) ==> node(A), node(B).
```

The idea is to allow the creation of edges without having to add node constraints first (cf. example 15). Unfortunately, this innocent-looking rule causes unexpected behavior. Consider the query "`edge(a,b), edge(b,c)`". The first constraint propagates `node(a)` and `node(b)` because of `add_nodes`. After activating the second constraint a duplicate constraint `node(b)` will be activated, *before* `node(c)` is added to the store. The former constraint is redundant so it is removed by `set_sem_nodes`. This removal however triggers `missing_to`, which deletes `edge(b,c)`. Remember, there is no `node(c)` yet! So the final store contains `edge(a,b),node(a),node(b),node(c)` but not `edge(b,c)`. One way to solve this problem is to replace `add_nodes` with:

```
edge(A, _) \\ node(A) ==> node(A) pragma passive(node/1).
edge(_, B) \\ node(B) ==> node(B) pragma passive(node/1).
```

We have extended the `passive` pragma [13] to negated heads: rules do not trigger on the removal of negated heads that are declared `passive` (see [20]). Here the pragmas avoid making node removal impossible for non-isolated nodes.

The above example shows two issues:

- There is a disparity between the two phases of a rule application: constraints are removed from the store *atomically* (removed heads are removed together) while they are added *incrementally* (one constraint at a time).
- Removing a constraint (e.g. `node(b)`) can trigger negated heads that test on the absence of other constraints (`node(c)`).

In fact, most issues raised by the above examples are symptoms of the same, fundamental problem of integrating negation as absence in CHR while preserving the conventional ω_r semantics. In ω_r, constraints (from a query or a rule body) are added to their respective stores (user-defined or built-in) one at a time, treating each addition as a procedure call (cf. Section 3). The problem is that each of these constraint calls will look for applicable rules disregarding the constraints that might be added shortly. Each applicable rule found is fired, and each of the constraints in their body will *in turn* have no knowledge of any unactivated constraints lower on the execution stack. This is a known issue with the common operational semantics of regular CHR, but it hardly ever poses a problem in practice. However, as shown by the above examples, the integration of negation severely exacerbates the problem.

Potential solutions. Our preliminary attempts to formulate a clean logical reading for $\overline{\text{CHR}}$ have failed in both classical and linear logic. However, it may be possible to obtain a declarative semantics in a non-monotonic logic [3] like *Transaction Logic* ($\mathcal{TR}$) [14].

Additional pragmas could be added to avoid some of the practical problems shown above, but this approach is ad-hoc and seems contrary to the declarative nature of CHR. A more satisfactory solution would be based on a significantly different operational semantics for CHR (and $\text{CHR}^{\neg}$) — e.g. a more breadth-first batch-like semantics or one in which the rule order is enforced more strongly than in ω_r. There has been little interest in an alternative operational semantics for CHR in the past, but it seems to be a promising area of future research.

5 Conclusion

We introduced $\text{CHR}^{\neg}$, an extension of CHR with negation as absence. $\text{CHR}^{\neg}$ rules can test for constraint *absence* and can fire after constraint *removal*. We proposed an operational semantics $\omega_r^{\neg}$ for $\text{CHR}^{\neg}$, and formalized it as an extension of the regular CHR semantics. We illustrated the benefits and issues of $\omega_r^{\neg}$ and some of the decisions made in designing it. We developed a prototype implementation based on a source-to-source transformation form $\text{CHR}^{\neg}$ to CHR [20].

The general conclusion of this exploratory work is one of mixed feelings. On the one hand, many beautiful theoretical properties of CHR are lost in $\text{CHR}^{\neg}$. The issues discussed in Section 4 also indicate that programming in $\text{CHR}^{\neg}$ might require a thorough understanding of the execution mechanism. This is inconsistent with the declarative nature of CHR. On the other hand, negation increases the expressiveness and conciseness of rules, and adds a nice symmetry between the operational semantics of constraint insertion and constraint removal.

Related Work. Other, related languages offer features similar to negation as absence. The rules proposed in [22] — used to update (incomplete) knowledge bases — can contain both classical negation and negation as absence. In the Petri Net area, *inhibitor arcs*, testing for the absence of tokens, are long known [5]. All major production rule systems [8, 10, 15, 16] provide negation as absence. In fact, most [10, 15, 16] allow the logical connectives `and`, `or` and `not` — arbitrary nested — on the left-hand side of their rules. In [7], an extension of production rules with *negation as failure* is motivated and explored. A more detailed overview of related work can be found in [20]. A common thread is that rules are applied atomically [9], which relaxes the practical issues discussed in Section 4.

In the context of CHR, [1] proposed the technique of *explicit negation*, which amounts to adding, for every constraint `c/n`, an auxiliary constraint `notc/n` and a rule of the form `c(`$\bar{X}$`), notc(`$\bar{X}$`) ==> fail`. This allows queries and rule bodies to *tell* the *classical* negation of a constraint. To the best of our knowledge, we are the first to examine negation *as absence* in the context of CHR.

Future Work. We expect $\text{CHR}^{\neg}$ to have interesting but complicated theoretical and practical properties. Our work can be seen as a proposal for — and an early experiment in — intensified research on negation and CHR. Section 4 already indicated several possible research directions in the context of negation

as absence. Other forms of negation can be investigated as well. In time, an optimized compilation of CHR$^{\neg}$ might be warranted. Other left-hand side extensions (disjunction, nested logical operators, ...) can also be explored.

Acknowledgements. We thank Thom Frühwirth and the members of his CHR research team, Hariolf Betz, Martin Käser, Marc Meister and Jairson Vitorino, for the interesting and relevant discussions during their recent visit to our department. We are also grateful to the anonymous referees for their helpful comments.

References

1. S. Abdennadher and H. Christiansen. An experimental CLP platform for integrity constraints and abduction. In *Intl. Conf. Flexible Query Answering Systems*, 2000.
2. S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal*, 4(2):133–165, 1999.
3. G. A. Antonelli. Non-monotonic logic. Stanford Encyclopedia of Philosophy, 2006. http://plato.stanford.edu/archives/spr2006/entries/logic-nonmonotonic/.
4. H. Betz and T. Frühwirth. A linear-logic semantics for CHR. In *11th Intl. Conf. Principles and Practice of Constraint Programming*, 2005.
5. S. Christensen and N. D. Hansen. Coloured Petri nets extended with place capacities, test arcs and inhibitor arcs. In *Appl. and Theory of Petri Nets*, 1993.
6. G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. The refined operational semantics of CHR. In *20th Intl. Conf. Logic Programming*, 2004.
7. P.M. Dung and P. Mancarella. Production systems with negation as failure. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):336–352, 2002.
8. C. L. Forgey. OPS5 User's Manual. Technical Report CMU-CS-81-135, Carnegie Mellon University, Dept. CS, 1981.
9. C. L. Forgey. RETE: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.
10. E. Friedman-Hill et al. Jess home page. http://www.jessrules.com/, June 2006.
11. T. Frühwirth. Theory and practice of CHR. *J. Logic Programming*, 37, 1998.
12. T. Frühwirth. Parallelizing union-find in CHR using confluence analysis. In *21st Intl. Conf. Logic Programming*, 2005.
13. C. Holzbaur and T. Frühwirth. CHR reference manual for SICStus Prolog. Technical Report TR-98-01, Austrian Research Institute for Artificial Intelligence, 1998.
14. M. Meister. A Transaction Logic Semantics for CHR. Seminar Day on CHR (http://www.cs.kuleuven.be/∼toms/CHR/chr_wog.html), May 2006.
15. M. Proctor et al. JBoss Rules. http://www.jboss.com/products/rules, June 2006.
16. G. Riley et al. CLIPS home page. http://www.ghg.net/clips/CLIPS.html, 2006.
17. T. Schrijvers. K.U.Leuven CHR system. http://www.cs.kuleuven.be/∼toms/CHR.
18. J. Sneyers et al. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In *20th Workshop on Logic Programming*, 2006.
19. J. Sneyers, T. Schrijvers, and B. Demoen. Memory reuse for CHR. In *22nd Intl. Conf. Logic Programming*, August 2006. To appear.
20. P. Van Weert, J. Sneyers, et al. To CHR$^{\neg}$ or not to CHR$^{\neg}$: Extending CHR with negation as absence. Technical Report CW 446, K.U.Leuven, Dept. CS, 2006.
21. P. Van Weert, T. Schrijvers, and B. Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In *2nd Workshop on CHR*, 2005.
22. Y. Zhang and N.Y. Foo. Towards generalized rule-based updates. In *15th Intl. Joint Conference on Artificial Intelligence*, volume 1, pages 82–88, August 1997.

Translating Constraint Handling Rules into Action Rules

Tom Schrijvers[2⋆], Neng-Fa Zhou[1] and Bart Demoen[2]

[1] Department of Computer & Information Science
CUNY Brookly College & Graduate Center
zhou@csi.brooklyn.cuny.edu
[2] Department of Computer Science
K.U.Leuven, Belgium
{toms,bmd}@cs.kuleuven.be

Abstract. CHR is a popular high-level language for implementing constraint solvers and other general purpose applications. It has a well-established operational semantics and quite a number of different implementations, prominently in Prolog. However, there is still much room for exploring the compilation of CHR to Prolog. Nearly all implementations rely on attributed variables. In this paper, we explore a different implementation target for CHR: B-Prolog's Action Rules (ARs). As a rule-based language, it is a good match for particular aspects of CHR. However, the strict adherence to CHR's refined operational semantics poses some difficulty. We report on our work in progress: a novel compilation schema, required changes to the AR language and the preliminary benchmarks and experiences.

1 Introduction

Constraint Handling Rules (CHR) [8] is a rule-based programming language commonly embedded in a host language. It is a powerful yet relatively simple programming language that combines elements of Constraint (Logic) Programming (CLP) and rule-based languages. CHR's is intended as a language for implementing user-defined application-tailored constraint solvers, but it is also used as a general programming language.

Several implementations of CHR exist and most are embedded in Prolog [10, 12, 15] and HAL [5, 11]. CHR implementations for Java [1, 18] and Haskell [17] exist as well. The implementation [10] in SICStus Prolog is generally considered the reference implementation because it is historically the first full-fledged CHR system. It implements an efficient compilation schema in terms of attributed variables. More recently, the formulation of the refined operational semantics of CHR [6] has captured the essentials of the reference implementation on a more formal level.

AR (Action Rules) is a rule-based event-handling language originally designed for programming constraint propagation [19]. It has been employed in

⋆ Research Assistant of the fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

the implementations of several highly efficient constraint solvers. There are two major differences between AR and CHR: (1) CHR allows multi-headed rules while AR only accepts single-headed rules; and (2) unlike in CHR where primitive delay conditions are hidden in heads and guards of rules, all primitive delay conditions must be explicit in AR. It is not obvious how CHR can be compiled into AR, a seemingly lower-level language.

The goal of this paper is to explore a novel Prolog implementation of CHR in terms of Action Rules. Traditionally, CHR is compiled to Prolog using attributed variables [9], a low-level primitive for implementing constraint solvers. We argue that because of its rule-based nature, higher level of expressivity, Action Rules may provide a fair alternative compilation target. In our exploration we hope to assess this claim and gain new insights into the efficient compilation of CHR.

The rest of this paper is structured as follows. In the next section, we provide a brief overview of CHR-related concepts that are important for this paper. Subsequently, in Section 3 we introduce the Action Rules language. The novel compilation schema from CHR to AR is presented in two steps. Firstly, in Section 4, as basic compilation schema is presented. Secondly, a number of optimizations to this basic schema are sketched in Section 5. We report on a preliminary experimental evaluation in Section 6. Finally, in Section 7 we summarize and discuss our experiences.

There is a companion technical report [16] for this paper, that contains more elaborate explanation, full compilation schemas and optimizations, which we could not include in this text for lack of space.

2 Constraint Handling Rules

We assume the reader to be already familiar with the general aspects of the CHR language, its syntax and operational semantics.

The compilation schema we will present, implements the refined operational semantics [6], which is the de facto standard operational semantics of CHR implemented by all major CHR systems.

An important notation in the refined semantics, is the *occurrence* of a constraint symbol and the order of occurrences. The order is textual following the rules, and in a head it is from left to right, except for the simpagation rule, where occurrences to the right of the \ comes before the occurrences to the left.

The refined semantics follows a depth first execution strategy. One constraint is activated at a time and for this constraint the occurrences are visited in order. At each occurrence, the corresponding rule is attempted. If the rule succeeds, execution of the active constraint is temporarily suspended, in favor of depth first execution of the rule body. For further details we refer to [6].

3 AR: Action Rules

The AR (*Action Rules*) language is designed to facilitate the specification of event-driven functionality needed by applications such as constraint propagators

and graphical user interfaces where interactions of multiple entities are essential [19]. It has been used to implement the efficient finite-domain and finite-set domain solvers in B-Prolog.

An action rule takes the following form: "$H, G, \{E\}=>B$" where H (called the *head*) is an atomic formula that represents a pattern for agents, G (called the *guard*) is a conjunction of conditions on the agents, E (called *event patterns*) is a non-empty disjunction of patterns for events that can activate the agents, and B (called *action*) is a sequence of arbitrary subgoals. An action rule degenerates into a *commitment rule* if E together with the enclosing braces are missing. In general, a predicate can be defined with multiple action rules. For the sake of simplicity, we assume in this paper that each predicate is defined with only one action rule possibly followed by a sequence of commitment rules.

For this paper, we are interested in the following event patterns:

- `generated`: After an agent is generated but before it is suspended for the first time. The sole purpose of this event pattern is to make it possible to specify preprocessing and constraint propagation actions in one rule.
- `ins`(X): when the variable X is instantiated.
- `event`(X, T): the general form of user-defined events, where X is a variable, called a *channel*, and T is a Prolog term, called an *event object*.

In general, an action rule may specify several event patterns such as `ins` patterns on several variables.

For an agent, a rule is *applicable* to it if the agent matches the head of the rule and the guard is satisfied. When an agent is created, the system checks if the action rule in its predicate is applicable to it.[3] If so, the agent will be suspended until it is *activated* by one of the events specified in the rule. Whenever the agent is activated by an event, the condition of the action rule is tested *again*. If it is met, the action is executed. The agent does not vanish after the action is executed, but instead sleeps until it is activated again. There is no primitive for killing agents explicitly. An agent vanishes only when a commitment rule is applied to it.

Events are posted though a channel to agents by the system or by the user program. The built-in primitive `post_event`(C, T) posts a general event to the agents connected to C, where C is a *channel*.The activated agents are first connected to the active chain of agents and are then executed one at a time. Therefore, agents are activated in a *breadth-first* fasion. The second argument can be omitted if no information needs to be transmitted to the activated agents.

There is no primitive for killing agents explicitly. As described above, an agent never disappears as long as action rules are applied to it. An agent vanishes only when a commitment rule is applied to it. Consider the following example.

```
p(X,Flag), var(Flag), {event(X,O)} =>
        write(O),Flag=1.
p(X,Flag) => true.
```

[3] Since one-directional matching rather than full-unification is used to search for an applicable rule and only tests are allowed in the guard, the agent will remain the same after an applicable rule is found.

An agent defined here can only handle one event posting. After it handles an event, it binds the variable `Flag`. So, when a second event is posted, the action rule is no longer applicable and thus the commitment rule after it will be selected.

One question arises here: what happens if there will never be another event on `X`? In this case, the agent will stay forever. If we want to kill the agent immediately after it is activated once, we have to define it as follows:

```
p(X,Flag), var(Flag), {event(X,O),ins(Flag)} =>
        write(O),Flag=1.
p(X,Flag) => true.
```

In this way, the agent will be activated again after `Flag` is bound to 1, and be killed after the failure of the test `var(Flag)`.

4 The Basic Translation Schema

This section gives the general schema for translating CHR rules into AR rules. We focus on translating double-headed CHR rules. For single headed rules, a head constraint is treated as the partner of itself (discussed in Subsection 4.4), and rules with more than two constraint patterns in the head are translated to double-headed rule (discussed in Subsection 4.5) before translating them into AR rules.

4.1 Preparation

Without loss of generality we assume that all rules in the given CHR program have been transformed into the *head normal form* [11] where all matching operations are encoded explicitly in the guards of the rules. Hence, all the arguments of the constraint patterns in the program are unique variables.

For each constraint $p(\overline{X})$, an event of the following form is posted when the constraint is added into the store and when any variable in it is instantiated:

$$\texttt{constr}(Cno, Alive, History, \overline{X})$$

where Cno is a unique identifier of the constraint, $Alive$ is a logic variable, called a *status variable*, used to indicate aliveness of the constraint (the variable is free if the constraint is alive and is bound to 0 if it has been removed from the store), and $History$ is the firing history of propagation rules in which the constraint is involved. The data structure for the history is immaterial at this moment: it could very well be an open-ended list. We need the following primitives:

- `gen_constr_id`(Cno): Generates a new identifier Cno. The nature of the identifier is immaterial; an integer is used in our implementation.
- `not_in_history`($History, PartnerNo, RuleNo$):
- `add_to_history`($History, PartnerNo, RuleNo$): Let $History$ be the history associted with a constraint. The primitive `not_in_history` succeeds if the tuple $(PartnerNo, RuleNo)$ is not included in $History$, and the primitive `add_to_history` adds the tuple $(PartnerNo, RuleNo)$ into $History$.

For each occurrence of a constraint symbol in the heads, we use a unique identifier, called an *occurrence identifier*, to denote it, which is composed of the constraint symbol, the arity, and the number of the occurrence. For example. the fourth occurrence of a constraint symbol `p/1` has identifier `p_1_4`.

For each constraint symbol, we define two lists of occurrence identifiers: The *primal occurrence* list contains the occurrence identifiers of that symbol and the *partner occurrence* list contains the identifiers of the partner occurrences. The two lists overlap if there is a rule with two identical constraint symbols in the head. The ordering of primal occurrences is not important, but the occurrence identifiers in the partner list must be in the order specified by the refined operational semantics.

For each occurrence, a channel is used for posting and receiving events. For each instance in the store of the corresponding constraint symbol, there is an agent attached to the channel. We need the following primitive:

- `get_channel`$(Id, Channel)$: retrieves the channel for the occurrence with the identifier `Id`.[4]

A channel is assigned to an identifier when `get_channel` is called on it for the first time.

4.2 The basic schema for two-headed rules

There is one Prolog clause generated for each constraint symbol. Let p/n be a constraint symbol with the primal occurrence list $[P_1, \ldots, P_k]$ and the partner occurrence list $[Q_1, \ldots, Q_m]$. Our basic schema generates the following Prolog clause for the symbol:

$p(\overline{X}) :-$
`gen_constr_id`(Cno),
$Constr = constr(Cno, Alive, History, \overline{X})$,
`get_channel`(P_1, ChP_1),...,`get_channel`(P_k, ChP_k),
`get_channel`(Q_1, ChQ_1),...,`get_channel`(Q_m, ChQ_m),
`agent_`$P_1(ChP_1, Cno, Alive, History, \overline{X})$,
...
`agent_`$P_k(ChP_k, Cno, Alive, History, \overline{X})$,
`post_`$p\_n(ChQ_1, \ldots, ChQ_m, Alive, Constr, \overline{X})$.

For each occurrence identifier in the primal and partner occurrence lists, there is a call of `get_channel`, which gets the channel for the occurrence. For each primal occurrence P_i $(i = 1, \ldots, k)$, there is a call named `agent_`P_i, which creates an agent for waiting for future partner constraints. The last call in the clause `post_`p_n posts the constraint $p(\overline{X})$ to the channels of the partner occurrences to initiate search for partners of $p(\overline{X})$.

[4] The channel of an occurrence is an attributed variable stored as a global heap variable. In B-Prolog, the built-in `global_heap_get(Name,Value)`, which is the same as `b_getval(Name,Value)` in h-Prolog and SWI-Prolog, is used to access global heap variables.

Let the identifier of the rule in which P_i occurs be $RuleId$, the guard and body of the rule be G and B, respectively. If the rule is a propagation rule, then the agent for the occurrence P_i is defined as follows:

```
agent_Pi(Ch, Cno, Alive, History, X),
    var(Alive),
    {event(Ch, Q),ins(Alive)}
=>
    Q = constr(CnoQ, AliveQ, HistoryQ, Y),
    (Cno\==CnoQ,
    var(AliveQ)
     not_in_history(History, CnoQ, RuleId),
     not_in_history(HistoryQ, Cno, RuleId),
     G ->
         add_to_history(History, CnoQ, RuleId),
         add_to_history(HistoryQ, Cno, RuleId),
         B
     ;
         true).
agent_Pi(_, _, _, _, X) =>true.
```

When the agent receives an event $constr(CnoQ, AliveQ, HistoryQ, \overline{Y})$, it checks the following: (1) the constraint represented by the event is different from the constraint represented by this agent ($Cno\backslash==CnoQ$); (2) the constraint represented by the event is alive ($\texttt{var}(AliveQ)$); (3) the rule $RuleId$ has never been applied on $(Cno, CnoQ)$ or $(CnoQ, Cno)$; and (4) the guard G is satisfied. The body is executed when all these four conditions are satisfied. The histories associated with these two constraints are updated to record this application before the body is executed. Notice that the agent also watches the $\texttt{ins}(Alive)$ event. When the constrain represented by the agent is removed from the store ($Alive$ is bound to 0), the agent will be killed.

If the rule in which P_i occurs is a simplification rule, then the following action rule is generated for P_i:

```
agent_Pi(Ch, Cno, Alive, History, X),
    var(Alive),
    {event(Ch, Q),ins(Alive)}
=>
    Q = constr(CnoQ, AliveQ, HistoryQ, Y),
    (Cno\==CnoQ,
    var(AliveQ)
     G ->
         Alive = 0, AliveQ = 0,
         B
     ;
         true).
```

The status variables *Alive* and *AliveQ* are set to 0 to indicate that the constraints represented by the agent and the event have been removed from the store. If the rule is a simpagation rule, then only *Alive* or *AliveQ* is set to 0 depending on whether P_i occurs to the left or right of \.

After all the agents `agent_`P_1, ..., `agent_`P_k are created, an agent named `post_`p`_`n is created to initiate search for partners.

`post_`p`_`$n(ChQ_1, \ldots, ChQ_m, Alive, Constr, \overline{X})$
    `var`$(Alive)$,
    {`generated`,`ins`$(Alive)$,`ins`$(\overline{X})$}
=>
    `post_event`$(ChQ_1, Constr)$,
    ...
    `post_event`$(ChQ_m, Constr)$.
`post_`p`_`$n(ChQ_1, \ldots, ChQ_m, \_, \_, \overline{X})$ =>`true`.

When the constraint $p(\overline{X})$ is created (as indicated by the event pattern `generated`) or when any variable in it is instantiated (as indicated by the event pattern `ins`$(\overline{X})$), the event representing the constraint *Constr* is posted to the channels of the partner occurrences to initiate search for partners. The refined semantics is preserved by posting the constraint to the channels of the occurrences in the specified order.

4.3 An Example

Consider the following CHR rule:

```
prime(Y) \ prime(X) <=> 0 is X mod Y | true.
```

where `prime(X)` is numbered 1 and `prime(Y)` is numbered 2. The following shows the generated code.

```
prime(X) :-
        gen_constr_id(Id),
        Constr = constr(Id,Alive,History,X),
        get_channel(prime_1_1,Ch1),
        get_channel(prime_1_2,Ch2),
        agent_prime_1_1(Ch1,Id,Alive,History,X),
        agent_prime_1_2(Ch2,Id,Alive,History,X),
        post_prime_1(Ch1,Ch2,Alive,Constr,X).

agent_prime_1_1(Ch,Id,Alive,History,X), var(Alive),
        {event(Ch,Constr), ins(Alive)} =>
        Constr=constr(IdQ,AliveQ,HistoryQ,Y),
        (Id\==IdQ,
         var(AliveQ),
         0 is X mod Y ->
            Alive = 0
```

```
        ;
            true
        ).
agent_prime_1_1(_,_,_,_,_) => true.

agent_prime_1_2(Ch,Id,Alive,History,Y), var(Alive),
        {event(Ch,Constr), ins(Alive)} =>
        Constr=constr(IdQ,AliveQ,HistoryQ,X),
        (Id\==IdQ,
         var(AliveQ),
         0 is X mod Y ->
            AliveQ = 0
        ;
            true
        ).
agent_prime_1_2(_,_,_,_,_) => true.

post_prime_1(Ch1,Ch2,Alive,Constr,X), var(Alive),
        {generated, ins(Alive), ins(X)} =>
        post_event(Ch1,Constr),
        post_event(Ch2,Constr).
post_prime_1(_,_,_,_,_) => true.
```

4.4 Single-headed rules

For a single-headed CHR rule with the constraint symbol p/n in the head, a constraint of p/n does not need to wait for a partner constraint to trigger the rule. Let P_i be the occurrence identifier of the head. Each time a constraint of p/n is added into the store, an agent defined below is created:

$$\begin{array}{l}\texttt{agent\_}P_i(PrivateCh, Cno, Alive, History, \overline{X}),\\ \quad \texttt{var}(Alive),\\ \quad \{\texttt{event}(PrivateCh, \_),\texttt{ins}(Alive)\}\\ \Rightarrow \ldots\\ \texttt{agent\_}P_i(\_,\_,\_,\_,\overline{X}) \Rightarrow \texttt{true}.\end{array}$$

where the body of the action rule encodes the guard and body of the CHR rule. The agent is not activated by the posting of a partner constraint, but by the posting of the constraint itself. The event pattern $\texttt{event}(PrivateCh, \_)$ means that no information from any event is used. By using a separate channel for each occurrence of p/n, we enforce as before the ordering of the rules in the refined semantics. However, now we have to use a private channel (one for each constraint instance rather than a global one) to enforce the depth-first execution order of the refined semantics. Otherwise we would break our assumption that a constraint only features as its own partner for a single-headed rule. The result would be a kind of breadth-first execution order.

Consider the following CHR program, (partially) implementing boolean `and/3` and `not/2` constraints:

```
and(X,X,Z) <=> Z = X.
not(X,Y) \ and(X,Y,Z) <=> Z = 0.
```

The following shows the generated code for `and/3`.

```
and(X,Y,Z) :-
        gen_constr_id(Cno),
        Constr = constr(Cno,Alive,History,X,Y,Z),
        get_channel(and_3_2,PublicCh2),
        get_channel(not_2_1,ChNot),
        agent_and_3_1(PrivateCh1,Cno,Alive,History,X,Y,Z),
        agent_and_3_2(PublicCh2,Cno,Alive,History,X,Y,Z),
        post_and_3(PrivateCh1,ChNot,Alive,Constr,X,Y,Z).

agent_and_3_1(PrivateCh1,Cno,Alive,History,X,Y,Z), var(Alive),
        {event(PrivateCh1,_), ins(Alive)} =>
        (X == Y ->
            Alive = 0,
            Z = X
        ;
            true
        ).
agent_and_3_1(_,_,_,_,_,_,_) => true.

agent_and_3_2(PublicCh2,Cno,Alive,History,X,Y,Z), var(Alive),
        {event(PublicCh2,Constr), ins(Alive)} =>
        Constr=constr(Cno,AliveNot,HistoryNot,XNot,YNot),
        (Cno\==CnoNot,
         var(AliveNot),
         X == XNot,
         Y == YNot
          ->
            Alive = 0,
            Z = 0
        ;
            true
        ).
agent_and_3_2(_,_,_,_,_,_,_) => true.

post_and_3(PrivateCh1,ChNot,Alive,Constr,X,Y,Z), var(Alive),
        {generated, ins(Alive), ins(X), ins(Y), ins(Z)} =>
        post_event(PrivateCh1,Constr),
        post_event(ChNot,Constr).
post_and_3(_,_,_,_,_,_,_) => true.
```

4.5 Multi-headed rules

For rules with more than two constraint patterns in the heads, we apply a similar binarization technique as in the famous RETE algorithm [7] for production systems. This technique makes the intermediate joins of partner constraints manifest in temporary constraints.

Conceptually for a rule with $n \geq 2$ constraint patterns:

$$p_1(\overline{X}_1), \ldots, p_n(\overline{X}_n) ==> G \mid B.$$

We compile it into $(n-1)$ two-headed rules with $(n-1)$ new constraint symbols $\mathtt{p}_{1\ldots j}$ where $2 \leq j \leq (n-1)$:

$$\begin{array}{l} p_1(\overline{X}_1), p_2(\overline{X}_2) ==> p_{1..2}(\overline{X}_1, \overline{X}_2). \\ p_{1..2}(\overline{X}_1, \overline{X}_2), p_3(\overline{X}_3) ==> p_{1..3}(\overline{X}_1, \overline{X}_2, \overline{X}_3). \\ \vdots \\ p_{1..(n-1)}(\overline{X}_1, \ldots, \overline{X}_{(n-1)}), p_n(\overline{X}_n) ==> p_{1..n}(\overline{X}_1, \ldots, \overline{X}_n). \end{array}$$

It has often been claimed informally that the correct binarization of CHR rules can be expressed trivially as the above source-to-source transformation is correct. In effect, early implementations of CHR did not support any n-headed rules where $n > 2$, partially for that reason.

In reality, the binarization is not as simple as that, and we believe that it cannot be expressed fully as a source-to-source transformation. A number of issues arise that have to be dealt with at a lower level.

Firstly, no head constraint can be used twice in the matching against the constraint patterns in the original CHR rule. To ensure this, we let each temporary constraint carry the identifiers of the precedence constraints from which it is obtained. Each time before we fire the rule $p_{1..j}(\overline{X}_1, \ldots, \overline{X}_j), p_{j+1}(\overline{X}_{j+1}) ==> \ldots$, we check if $p_{j+1}(\overline{X}_{j+1})$ occurs in the precedence constraints that led to the generation of $p_{1..j}(\overline{X}_1, \ldots, \overline{X}_j)$. If so, the rule cannot be fired.

Secondly, the following relationship among the status variables must be maintained: whenever a precedence constraint is removed from the store, the temporary constraints created from it must be removed as well. For this reason we let each temporary constraint carry in a similar fashion the status variables of the precedence constraints. On an `ins` event of any of these status variables the agents for the temporary constraints are removed.

Thirdly, for simplification and simpagation rules, it should be possible at the end of the join chain, before executing the body of the original rule, to remove the original precedence constraints. For this purpose, the status variables of the precedence constraints must be carried by the temporary constraints. Luckily, this is already the case for the previous issue.

Finally, the temporary constraints should be properly synchronized with their precedence constraints. If a variable appearing in a precedence constraint is instantiated, any temporary constraint that caries that variable should not fire its event, before the events for the previous occurrences of the precedence constraint do. Our solution is to not fire events for temporary constraints upon `ins` events of the temporary constraints' variables. Instead every precedence constraint has

a private channel for each multi-headed occurrence. This channel is carried by the temporary constraints. Upon an `ins` event for the precedence constraint, first the usual events are posted for the preceding occurrences before an event is posted to this private channel. The temporary constraint react on this event by firing their usual events.

Consider the following articifial CHR program:

```
a(X) \ b(X), c(X) <=> X = 42.
```

The following shows the generated code for `a/1` and the temporary constraint `temp12/8`.

```
a(X) :-
        gen_constr_id(Cno),
        Constr = constr(Cno,Alive,History,X,ChTemp),
        get_channel(a_1_1,Ch1),
        get_channel(b_1_1,ChB),
        agent_a_1_1(Ch1,Cno,Alive,History,X,ChTemp),
        post_a_1(ChB,ChTemp,Alive,Constr,X,Y,Z).

agent_a_1_1(Ch,Cno,Alive,History,X,ChTemp), var(Alive),
        {event(Ch,Constr), ins(Alive)} =>
        Constr=constr(CnoB,AliveB,HistoryB,XB,ChTempB),
        (Cno\==CnoB,
         var(AliveB),
         not_in_history(History,CnoB,r1),
         not_in_history(HistoryB,Cno,r1)
          ->
            add_to_history(History,CnoB,r1),
            add_to_history(HistoryB,Cno,r1),
            temp12(X,XB,Cno,CnoB,Alive,AliveB,ChTemp,ChTempB)
        ;
            true
        ).
agent_a_1_1(_,_,_,_,_,_) => true.

post_a_1(ChB,ChTemp,Alive,Constr,X), var(Alive),
        {generated, ins(Alive), ins(X)} =>
        post_event(ChB,Constr),
        post_event(ChTemp,Constr).
post_a_1(_,_,_,_,_) => true.

temp12(XA,XB,CnoA,CnoB,AliveA,AliveB,ChTempA,ChTempB) :-
        gen_constr_id(Cno),
        Constr = constr(Cno,Alive,History,XA,XB,...),
        get_channel(temp12_1,Ch1),
        get_channel(c_1_1,ChC),
        agent_temp12_8_1(Ch1,Cno,Alive,History,XA,XB,...),
```

```
        post_temp12_8(ChC,Alive,Constr,AliveA,AliveB,ChTempA,ChTempB).

agent_temp12_8_1(Ch1,Cno,Alive,History,XA,XB,CnoA,CnoB,AliveA,AliveB,ChTempA,ChTemp
        var(Alive),var(AliveA),var(AliveB),
        {ins(Alive),ins(AliveA),ins(AliveB),event(CH1,Constr)} =>
        Constr=constr(CnoC,AliveC,HistoryC,XC,ChTempC),
        (Cno\==CnoC, CnoA\==CnoC, CnoB\==CnoC,
         var(AliveB),
         XA==XB, XB==XC
         ->
            AliveB=0, AliveC=0,
            XA = 42
        ;
            true
        ).
agent_temp12_8_1(_,_,_,_,_,_,_,_,_,_,_,_,_,_) => true.

post_temp12_8(ChC,Alive,Constr,AliveA,AliveB,ChTempA,ChTempB),
        var(Alive),var(AliveA),var(AliveB),
        {generated,ins(Alive),ins(AliveA),ins(AliveB),
         event(ChTempA,ChTempB} =>
        post_event(ChC,Constr).
post_temp12_8(_,_,_,_,_,_,_) => true.
```

5 Optimizations

The above compilation schema is quite basic in a number of ways. This section discusses a few optimizations that greatly improve the efficiency. We focus on significant adaptations of existing optimizations [11, 14] for ARs.

Indexing. In our basic schema the public channels used for finding partner constraints have no selectivity at all: all constraints with the appropriate constraint symbol are reached. Indexing greatly reduces the selectivity of partner lookups. In our AR schema we associate an index with a particular partner constraint lookup. In addition to the general channel for the lookup, one or more index channels may be provided. An index channel only reaches a constraint that has a particular variable or a particular term in some argument position. The occurrence agent of the constraint simply watches the additional index channels too. The post agent updates the constraint's index channels upon instantiation, and it looks for partner constraints through index channels if they are available.

If multiple index channels are available, these are combined using the new `post_event(Index1/\.../\Indexn,Event)` feature, that only posts the event to agents that are attached to *all* of the channels `Index1,...,Indexn`.

Single-headed rules. We can merge the agents of multiple consecutive single-headed rules and their corresponding private channels into a single agent and

a single private channel. The bodies of the rules are merged appropriately in a single body. Subsequent bodies of a propagation rule are put in conjunction (i.e. any continuation), and of simplification rules are put in the else-branch (i.e. fail continuation) of that body.

For a constraint symbol that is defined by only single-headed rules, no private channel is needed at all to control the ordering.

Never-triggering constraints. A constraint symbol whose instances do not *trigger*, e.g. because they are fully instantiated when called, does not require its agents to watch `ins/1` events of the arguments.

6 Evaluation

We have written an automatic translator from CHR to Action Rules, following the above basic compilation schema. This compiler is evaluated on a representative set of benchmarks from [13]. In addition, to get a good view on the attainable efficiency, we have further optimized the generated code of the smallest benchmarks by hand. The compiler, benchmarks and hand-optimize code are all available from the webpage `http://www.probp.com/chr/`.

As a reference for comparison, we take the K.U.Leuven CHR system [15] in SWI-Prolog. This is currently the most optimized CHR-system for Prolog. Tabel 1 lists the timings in milliseconds, obtained on a Pentium 4 2 GHz.

Benchmarks	B-Prolog		SWI-Prolog
	Basic Compiler	Hand-Optimized	K.U.Leuven CHR
`fib (22)`	38,333 (*)	246	3,210
`fulladder (6000)`	1,050	-	340
`leq (50)`	82,823	138	3,010
`leq2 (50)`	84,407	145	3,870
`primes (2500)`	3,049	1,350	6,990
`wfs (1000)`	11,783	-	2,920
`zebra (10)`	4,273	621	4,580

Table 1. Benchmark Timings

In order to interpret the figures in Tabel 1 correctly, one must take into account that B-Prolog is on average about 5 times faster than SWI-Prolog.

Unsurprisingly, the basic schema's performance is not so good. However, the hand-optimized code shows that drastic improvements are possible. In fact, such improvements are capable of considerably outperforming the K.U.Leuven CHR system in its current form. This suggests a worthwhile set of optimizations that can be generalized to other compilation schema, such as that of the K.U.Leuven CHR system.

In (*) we use the consulted rather than the compiled benchmark, because the garbage collector is not activated in the latter, causing a serious slowdown. We expect this to be fixed in a new release of B-Prolog. Furthermore, we cannot reliably experiment with multi-headed rules that involve **Reactivate** transitions yet. The Action Rules semantics currently allows for a delay between an instantiation and its corresponding `ins/1` event, where other code can be run in between that is out of order with respect to CHR's refined semantics. We will also address this issue in a future version.

7 Conclusion

In this paper we have presented a novel schema for compiling CHR to Action Rules. On the one hand, the AR schema has some elegance over the attributed variables schema in the expression as it allows some loops to be expressed implicitly, just like CHR. On the other hand, we have experienced some difficulty with strictly following the refined operational semantics of CHR; it is not such a good match for the operational semantics of AR.

Our initial experimental results show that the basic performance is rather bad, but hand-optimizations give very encouraging results and suggest worthwhile optimizations for all CHR systems. Another important result of our schema is that it shows how to properly binarize multi-headed rules, and that this is non-obvious. This shows the advantages of the attributed variables schema over the RETE-based approach. In [4] it was already shown that RETE's time and space behavior is worse than that of LEAPS [3], by which the attributed variables implementation of CHR was inspired.

A current limitation of the Action Rules schema is that it does not fully support an extension of CHR, called CHR$^\vee$ [2], which allows disjunctions in the bodies of rules. An Action Rule behaves as if the body is contained inside the ISO-Prolog `once/1` built-in: after executing the body, any remaining choice points created during the execution are pruned.

In future work, we will extend and automate the proposed optimizations. In particular, we will focus on Action Rules specific optimizations. A hybrid compilation schema seems another worthwhile topic for further research: selecting the most efficient parts of the attributed variables and Action Rules schemas, possibly based on properties of the program being compiled. This could also allow us to lift the current restriction of the Action Rule schema and fully support CHR$^\vee$. Finally, we will investigate were the strict adherence to the refined operational semantics can be lifted without harm.

References

1. Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauss. JACK: A Java Constraint Kit. In *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming*, Kiel, Germany, September 2001.
2. Slim Abdennadher and Heribert Schütz. CHR$^\vee$: a flexible query language. In *FQAS '98: Proceedings of the Third International Conference on Flexible Query Answering Systems*, pages 1–14, London, UK, 1998. Springer-Verlag.

3. Don Batory. The leaps algorithm. Technical report, Austin, TX, USA, 1994.
4. David A. Brant, Timothy Grose, Bernie Lofaso, and Daniel P. Miranker. Effects of database size on rule system performance: Five case studies. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 287–296. Morgan Kaufmann, 1991.
5. Bart Demoen, Maria García de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. An Overview of HAL. In Joxan Jaffar, editor, *CP'99: Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*, pages 174–188, Alexandria, Virginia, USA, 1999. Springer Verlag.
6. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *ICLP'04: Proceedings of the 20th International Conference on Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104, St-Malo, France, September 2004. Springer Verlag.
7. Charles L. Forgy. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19(1):17–37, 1982.
8. Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
9. Christian Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. Technical Report TR-92-23, Austrian Research Institute for Artificial Intelligence, Vienna, Austria, 1992.
10. Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules Compiler and Runtime System. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules*, 14(4), April 2000.
11. Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing Compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules*, 5(Issue 4 & 5):503–531, 2005.
12. IC-Parc. ECL^{i}PS^e. http://www.icparc.ic.ac.uk/eclipse/.
13. Tom Schrijvers. A Collection of Assorted CHR Benchmarks, 2005. http://www.cs.kuleuven.be/~toms/Research/CHR/.
14. Tom Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules*. Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium, June 2005. xxiv+210 pages.
15. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, Ulm, Germany, May 2004.
16. Tom Schrijvers, Neng-Fa Zhou, and Bart Demoen. The compilation schema for translating chr into action rules. Report CW 449, K.U.Leuven, Department of Computer Science, June 2006.
17. Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. *ACM Transactions on Programming Languages and Systems*, 2005. To appear.
18. Armin Wolf. Adaptive Constraint Handling with CHR in Java. In *CP'01: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science 2239, page 256. Springer Verlag, January 2001.
19. Neng-Fa Zhou. Programming finite-domain constraint propagators in action rules. *To appear in Theory and Practice of Logic Programming (TPLP)*, 2006.