Proceedings of CHR 2005, Second Workshop on Constraint Handling Rules

Tom Schrijvers *Thom Frühwirth*

Report CW 421, September 2005



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

Proceedings of CHR 2005, Second Workshop on Constraint Handling Rules

Tom Schrijvers *Thom Frühwirth*

Report CW 421, September 2005

Department of Computer Science, K.U.Leuven

Abstract

This book contains the Proceedings of CHR 2005, the Second Workshop on Constraint Handling Rules, held at the occasion of ICLP 2005 in Sitges (Spain) on October 5, 2005.

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.

This workshop follows the First Workshop on Constraint Handling Rules was organized in May 2004 in Ulm, Germany. 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.

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

Preface

This book contains the Proceedings of CHR 2005, the Second Workshop on Constraint Handling Rules, held at the occasion of ICLP 2005 in Sitges (Spain) on October 5, 2005.

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.

This workshop follows the First Workshop on Constraint Handling Rules was organized in May 2004 in Ulm, Germany. 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.

Ten relevant papers were submitted to the workshop, all of which received at least two review and most even three. One paper was rejected and another withdrawn. Eight excellent papers on CHR were accepted to the workshop.

New to the CHR workshop this year is the Best Paper Award. The Program Committee selected a best paper from the eight accepted papers, based on its outstanding quality in both presentation and scientific contribution and for its impact on the field of CHR.

We are privileged to have a distinguished invited speaker: Martin Sulzmann (National University of Singapore) talking about systematic type system design with 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, the ICLP 2005 organizers and the workshop chair Hai-Feng Guo.

August 2005 Tom Schrijvers & Thom Frühwirth

Organization

Workshop Coordinators

Tom Schrijvers (Belgium) and Thom Frühwirth (Germany)

Program Committee

Slim Abdennadher, German University in Cairo, Egypt

Marco Alberti, Universita degli Studi di Ferrara, Italy

Henning Christiansen, Roskilde University, Denmark

Gregory J. Duck, Melbourne Universiy, Australia

Thom Frühwirth, Universität Ulm, Germany

Christian Holzbaur, University of Vienna, Austria

Tom Schrijvers, Katholieke Universiteit Leuven, Belgium

Armin Wolf, Fraunhofer FIRST, Germany

Additional Referees

Marc Meister, Universität Ulm, Germany

Jon Sneyers, Katholieke Universiteit Leuven, Belgium

Best Paper Award

The Computational Power and Complexity of Constraint Handling Rules,
Jon Sneyers, Tom Schrijvers, Bart Demoen.

Table of Contents

Invited Talk: Chameleon: Systematic Type System Design via Constraint Handling Rules
Martin Sulzmann 1

The Computational Power and Complexity of Constraint Handling Rules
Jon Sneyers, Tom Schrijvers, Bart Demoen 3

A Type System for CHR
Emmanuel Coquery, François Fages 19

A High Performance CHR$^\vee$ Execution Engine
Luis Menezes, Jairson Vitorino, Marcos Aurelio 35

K.U.Leuven JCHR: a User-Friendly, Flexible and Efficient CHR System for Java
Peter Van Weert, Tom Schrijvers, Bart Demoen 47

Constraint Abduction and Constraint Handling Rules
Martin Sulzmann, Jeremy Wazny, Peter J. Stuckey 63

Logical Rules for a Lexicographic Order Constraint Solver
Thom Frühwirth 79

Reasoning about passive declarations in CHR
Henning Christiansen 93

Accurate Functional Dependency Analysis for Constraint Handling Rules
Gregory J. Duck, Tom Schrijvers 109

Chameleon: Systematic Type System Design via Constraint Handling Rules

Martin Sulzmann

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

Invited Talk

Type systems are important tools in the design, analysis, and verification of programming languages. Sophisticated applications often demand ad-hoc extensions to existing systems.

In this talk, I will give an overview of the main principles and ideas behind the Chameleon system. It integrates sophisticated reasoning capabilities into a programming language via its CHR programmable type system. Thus, we can program novel type system applications in terms of CHRs which previously required special-purpose systems. Furthermore, we can identify criteria in terms of CHRs under which we establish important properties such as decidability of type inference etc.

The Computational Power and Complexity of Constraint Handling Rules

Jon Sneyers*, Tom Schrijvers**, Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium

Abstract. We introduce the CHR machine, a model of computation based on the Constraint Handling Rules programming language. Its computational power and time complexity properties are compared with those of the Turing machine and Random Access Memory machine. Our major result is the proof that every algorithm can be implemented in CHR with the best known time and space complexity.

1 Introduction

Constraint Handling Rules (CHR) [5] is a high-level programming language extension based on multi-headed committed-choice rules. Originally designed for writing constraint solvers, it is increasingly used for general purposes.

After a new programming language is introduced, sooner or later the question arises whether classical algorithms can be implemented in an efficient and elegant way. In [15] this question is partially answered for CHR by describing a CHR implementation of the union-find algorithm. Thanks to some compiler optimizations, most notably hash-table constraint stores, the best known time complexity was obtained. In contrast, it is not clear whether the union-find algorithm can be implemented with optimal complexity in pure Prolog [7].

The major result of this paper is an affirmative answer to the first part of the above question: *is it possible, in general, to implement algorithms in CHR in an efficient way?* i.e. with the best known time and space complexity. We have investigated this question by introducing a new model of computation, the CHR machine, and comparing it with the Turing machine and RAM machine models.

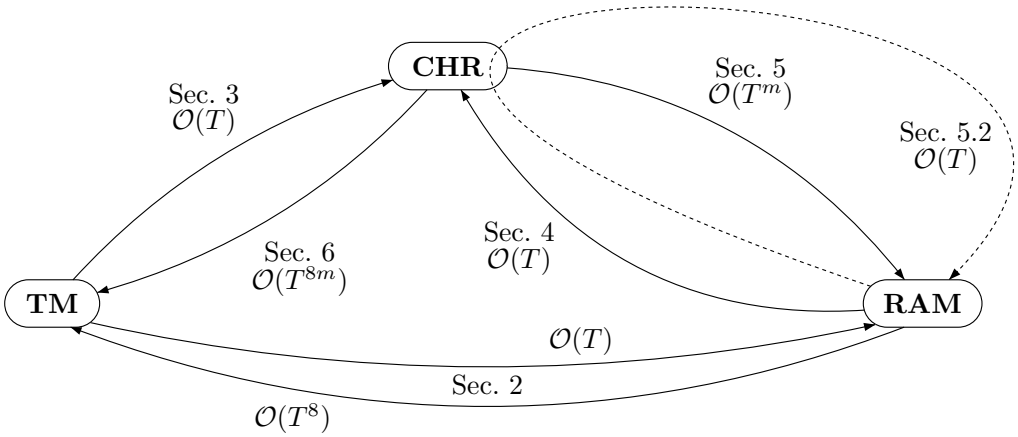
[6] gives a complexity meta-theorem for CHR programs with only simplification rules. It provides a general technique for obtaining (crude) upper bounds on the time complexity of such programs. Two rather orthogonal components need to be considered in this analysis: the number of rule applications in a derivation, and the cost of finding applicable rules. Considering the current state-of-the-art in CHR compilation, the former component depends only on the CHR program itself, while the latter is implementation-dependent. In an attempt to separate these two components, we will assume the CHR machine to be able to find applicable rules in constant time. This assumption is realistic for a class of CHR

* This work was partly supported by project G.0144.03 funded by F.W.O.-Vlaanderen.
** Research Assistant of the Research Foundation - Flanders (F.W.O.-Vlaanderen).

programs, including the simulators we will describe, but it is not feasible in the general case. New compiler optimizations allow a more efficient search for applicable rules. In our terminology, such optimizations correspond to better RAM machine simulators of the CHR machine.

Overview. In the next section we define Turing machines and RAM machines, because multiple definitions are used in the literature, and we introduce the CHR machine. We discuss simulation of Turing machines and RAM machines by CHR in Sections 3 and 4. Then, in Section 5, we investigate RAM machine simulation of CHR. From these results, we derive a theoretical simulation of CHR by a Turing machine in Section 6. Finally, in Section 7 we conclude this paper.

The following graph illustrates the global structure of this paper, and summarizes the complexity results. An arrow from computation model X to computation model Y means that X can be simulated in Y. The arrows are annotated with a reference to the relevant section, and an upper bound on the complexity penalty for the simulation, where T represents the original time complexity and m is the maximal number of head constraints in a rule of the CHR program.



2 Models of computation

In this section, we will describe three models of computation: the Turing machine, the Random Access Memory (RAM) machine, and the Constraint Handling Rules (CHR) machine. We discuss two variants of the RAM machine with a different instruction set: the Peano-arithmetic RAM machine and the standard-arithmetic RAM machine. Similarly, we define two variants of the CHR machine: the CHR-only machine and the minimal host-language CHR machine.

2.1 Turing machines

We will use the single-tape formulation of the Turing machine (TM). Multi-tape Turing machines can be simulated on single-tape Turing machines with a quadratic complexity penalty [8]. The definition below corresponds to [11].

Definition 1 (Turing Machine). *A Turing machine is defined as a 6-tuple* $M = \langle Q, \Sigma, s_0, b, F, \delta \rangle$ *where*

- Q *is a finite set of* states
- Σ *is a finite set of symbols, the tape* alphabet
- $s_0 \in Q$ *is the initial state*
- $b \in \Sigma$ *is the* blank *symbol*
- $F \subseteq Q$ *is the set of* accepting *final states*
- $\delta : Q \setminus F \times \Sigma \mapsto Q \times \Sigma \times \{L, R\}$ *is a partial function called the* transition function *where* L *is left shift,* R *is right shift*

In the literature definitions sometimes differ, but without an impact on the computational power or time complexity. For example the set $\{L, R\}$ is often extended with S, allowing the machine to stay on the same tape cell.

A Turing machine M operates on an infinite tape of cells. Each cell contains a symbol of the tape alphabet Σ. The tape is assumed to be arbitrarily extendible to the left and the right. A head is positioned on a particular cell of the tape, can read and write a symbol in that cell and can move left and right.

Operation starts in the initial state s_0 on a tape with a finite number of non-blank cells (called the *input*) and the head is positioned on the left-most non-blank symbol. Execution proceeds by considering the current state s and the symbol σ that is under the head. Then:

- Either (s, σ) is a member of the domain of δ and $\delta(s, \sigma) = (s', \sigma', X)$. The effect then is that the current state of M changes to s', the head overwrites the value σ in the cell under it with σ' and next the head either moves to the left or the right depending on whether $X = L$ or $X = R$.
- Or (s, σ) is not part of the domain of δ. Execution stops. If $s \in F$, the input is *accepted*, otherwise it is *rejected*.

Time complexity. The (worst-case) time complexity of a Turing machine M is the maximal number of execution steps, or state changes, as a function of the size of the input. The size of the input is usually defined as the number of symbols between the left-most and right-most non-blank symbols.

Turing-completeness. A programming language is *Turing-complete* if it has the same computational power as Turing machines. In other words, every Turing Machine can be simulated in the programming language and every program of the language can be simulated on a Turing machine.

2.2 Random Access Memory machines

The Random Access Memory (RAM) machine closely models the basic features of traditional sequential computers. In the literature many variations of the RAM have been considered. We investigate two different RAM machines. The first is a RAM machine with simple Peano arithmetic operations and the second has the standard arithmetic operations as they are implemented on today's computers.

Instruction		*Effect*
inc	A	Increment the value of register A by one.
dec	A	Decrement the value of register A by one.
clr	A	Set the value of register A to zero.
const	B A_1	Set the value of register A_1 to the value B.
add	A_2 A_1	Add the value of register A_2 to the value of register A_1.
sub	A_2 A_1	Subtract the value of register A_2 from the value of register A_1.
mult	A_2 A_1	Multiply the value of register A_2 to the value of register A_1.
div	A_2 A_1	Divide the value of register A_1 by the value of register A_2.
move	A_2 A_1	Set the value of register A_1 to the value of register A_2.
i_move	A_2 A_1	Set the value of register A_1 to the value of the register given by the value of register A_2.
move_i	A_2 A_1	Set the value of the register given by the value of register A_1 to the value of register A_2.
jump	L	Set the program counter to L.
cjump	A L	If the content of register A is zero, set the program counter to L; otherwise continue.
halt		Halt execution of the RAM.

Table 1. Instruction set of the RAM

Common architecture. A RAM machine consists of two components: the central processing unit (CPU) and a random-access memory. The memory consists of a number of cells, or registers. Every cell has an address and a value. Addresses are successive natural numbers. In our models the memory is split into two disjoint parts with separate address spaces: the first is the working memory whose cells contain integers and the second is the program memory whose cells contain the successive instructions of the program. The working memory is initialized appropriately, i.e. the values of all the memory cells are zero except those that contain the input parameters. By convention, particular memory cells may contain the output of an execution. We use $A, A_1, A_2, \ldots$ to represent working memory addresses and $L, L_1, \ldots$ for program memory addresses.

The CPU follows a fetch-and-execute cycle. It has a program counter that is initialized to the first program memory address. This program pointer contains the address of the program memory cell with the next instruction to be executed. The CPU fetches the instruction in the cell and performs the corresponding operations. This involves setting the program counter to the next instruction, by default the successor of the current address. Table 1 lists the instructions supported by the standard-arithmetic RAM machine. The Peano-arithmetic RAM machine uses a subset of these instructions.

Definition 2 (Peano-Arithmetic RAM). *A Peano-arithmetic RAM machine consists of a program and a working memory as described above. The program instructions are* ***inc****,* ***dec****,* ***clr****,* ***jump****,* ***cjump****, and* ***halt*** *(see Table 1).*

This corresponds to the definition given in [12]. All copying, addition and subtraction has to be done by repeated use of the **inc** and **dec** instructions. This makes the Peano-arithmetic RAM less realistic, as actual computers do provide instructions for addition and subtraction. However, pure CHR without built-in constraints does not provide any arithmetic functionality either. Hence, arithmetic must be encoded, for example using Peano arithmetic.

Definition 3 (Standard-Arithmetic RAM). *A standard-arithmetic RAM machine (or standard RAM) consists of a program and a working memory as described above. The program instruction set is given in Table 1.*

This instruction set is similar to the one defined in [3], and resembles more closely actual computers. The **inc**, **dec**, and **clr** instructions are redundant: they can be implemented using **add**, **sub**, and **const**.

Time complexity. Every fetch-and-execute cycle in the RAM machine comprises one step. For this to be realistic, the memory cells have to be of bounded size. Without upper bound, a standard RAM can compute, in $\mathcal{O}(n)$ steps and without input, numbers as high as 2^{2^n}, which require $\mathcal{O}(2^n)$ Turing machine steps just to write to the tape. This means that a polynomial relationship between standard RAM machines and Turing machines would be impossible. Addition and subtraction take linear time in the Peano-arithmetic RAM, whereas they happen in constant time in the standard RAM.

Turing-completeness of the RAM. The two formulations of the RAM machine are both Turing-complete. For both RAM machines, a T-time Turing machine can be simulated in $\mathcal{O}(T)$ time. The main difference between the two RAM machines is in the effect on time complexity when they are simulated on a Turing machine. In [12] it is reported that a T-time Peano-arithmetic RAM using S registers can be simulated on a Turing machine in $\mathcal{O}(ST \log^2 S)$ time. The standard RAM is also polynomially related to the Turing machine, although the complexity penalty of simulating a it on a TM is much higher. It can be shown (see e.g. [3]) that a standard RAM M with time complexity T can be simulated on a multi-tape TM with $\mathcal{O}(T^4)$ time complexity[1]. Because simulating a multi-tape TM on a single-tape TM squares the time complexity [8], we have:

Lemma 1. *Any standard-arithmetic RAM machine with time complexity T can be simulated on a Turing machine with time complexity $\mathcal{O}(T^8)$.*

2.3 CHR machines

We assume the reader to be familiar with CHR. See [5] for an overview of the Constraint Handling Rules (CHR) language. We say a CHR program is *CHR-only* (this term was coined in [14]) if its rules do not contain any host-language

[1] Actually, a tighter bounds of $\mathcal{O}\big((T \log T \log\log T)^2\big)$ may be shown, as mentioned in a footnote on page 26 of [3].

built-ins in the body, and contain only syntactic (in)equalities in the guard. Furthermore, no compound Prolog terms are allowed: constraint arguments are constants or variables.

If we assume the existence of constant-time built-in arithmetic operations (one-way constraints) provided by the host language, then CHR can match the arithmetic efficiency of the standard RAM. This assumption is not unrealistic as the host language of CHR is often a language implemented on a standard RAM. For example, the host language Prolog provides the `is/2` one-way constraint that implements the arithmetic operations through delegation of the corresponding operations to the RAM on which Prolog is interpreted. Therefore, it makes sense to look at CHR programs in which statements are allowed that are likely to be available in every host-language: the basic arithmetic and comparison operations. We call such programs *minimal host-language CHR programs*. The set of extra built-in constraints must contain at least $+$, $-$, $*$, $/$, $=$, and $\neq$.

Operational semantics. We will use the standard operational semantics of CHR, sometimes also called *abstract* or *high-level* operational semantics. Several versions of the operational semantics have already appeared in the literature, e.g. [1, 5]. We adopt the version of [4], which is equivalent to the previous ones, but closer in formulation to the refined semantics, used in most implementations. The operational semantics is formulated as a state transition system.

Execution state. Firstly, we define an *execution state* σ as a tuple $\langle G, S, B, T\rangle_n$. The first part of the tuple, the *goal* G, is the multiset of constraints to be rewritten to solved form. The CHR constraint *store* S is the multiset of *identified* CHR constraints that can be matched with rules in the program P. An *identified* CHR constraint is a CHR constraint c associated with some unique integer i, which serves to differentiate among copies of the same constraint. We introduce the functions $chr(c\#i) = c$ and $id(c\#i) = i$, and extend them to sequences, sets and multisets of identified CHR constraints in the obvious manner. The constraint theory of the built-in constraints is denoted by $\mathcal{D}_b$.

The *built-in constraint store* B is the conjunction of all built-in constraints that have been passed to the underlying solver. For CHR-only programs, the built-in store is not needed. Since we will usually have no information about the internal representation of B, we will model it as a logical conjunction of constraints. The *propagation history* T is a set of sequences, each recording the identities of the CHR constraints that fired a rule, and the name of the rule itself. This is necessary to prevent trivial non-termination for propagation rules: a propagation rule is allowed to fire on a set of constraints only if the constraints have not been used (in the same order) to fire the same rule before. Finally, the counter n represents the next free integer that can be used to number a CHR constraint. Given an input query Q, the *initial execution state* is: $\langle Q, \emptyset, true, \emptyset\rangle_1$.

Transition rules. The standard operational semantics is based on the three transition rules listed in Figure 1. The first rule tells the underlying solver to

1. Solve $\langle \{c\} \uplus G, S, B, T\rangle_n \rightarrowtail_{solve} \langle G, S, c \wedge B, T\rangle_n$ where c is a built-in constraint. This transition does not occur in CHR-only programs.

---

2. Introduce $\langle \{c\} \uplus G, S, B, T\rangle_n \rightarrowtail_{introduce} \langle G, \{c\#n\} \uplus S, B, T\rangle_{(n+1)}$ where c is a CHR constraint.

---

3. Apply $\langle G, H_1 \uplus H_2 \uplus S, B, T\rangle_n \rightarrowtail_{apply} \langle C \uplus G, H_1 \uplus S, \theta \wedge B, T \cup \{h\}\rangle_n$ where there exists a (renamed apart) rule in $\mathcal{P}$ of the form

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

and a matching substitution θ such that $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$ and $\mathcal{D}_b \models B \rightarrow \exists_B(\theta \wedge g)$ and $h = (r, id(H_1), id(H_2)) \notin T$.

Fig. 1. The transition rules of the standard operational semantics

add a new built-in constraint to the built-in constraint store. Of course, this transition rule is not relevant for CHR-only programs. The second adds a new identified CHR constraint to the CHR constraint store. The last one chooses a program rule for which matching constraints exist in the CHR constraint store, and whose guard is entailed by the underlying solver, and *fires* it.

The transitions are non-deterministically applied, starting from the initial execution state, until either no more transitions are applicable (a successful derivation), or the underlying solver can prove $\mathcal{D}_b \models \neg\exists_\emptyset B$ (a failed derivation). In both cases a *final state* has been reached, the former a *success state* and the latter a *failed state*. CHR-only programs do not have failed derivations.

Definition 4 (CHR-only Machine). *A* CHR-only machine *consists of a CHR-only program and an execution state. Its initial state is given by an* input query*, which is a sequence of constraints. The machine works by exhaustively applying the transitions of the operational semantics, given in Figure 1. The output, or* solved form*, is given by the remaining constraints in the constraint store.*

Definition 5 (Minimal Host-language CHR Machine). *A* minimal host-language CHR machine *consists of a minimal host-language CHR program and an execution state. Input, processing, and output are as in definition 4.*

Clearly, the CHR-only machines are a subset of minimal host-language CHR machines. Both machines are theoretical models because we will assume they can do every transition in constant time.

Time complexity. The time complexity of a particular CHR derivation from an initial to a final state is the number of derivation steps. For a given CHR program P, the time complexity $f_P(q)$ of an initial query q is the maximum length of all derivations starting from q, or $+\infty$ if one of these derivation does not terminate. The size $|q|$ of a query q is the number of constraints in the query.

Definition 6. *The time complexity of a CHR program P is a function* $\text{time}_P(n)$*:*

$$\text{time}_P(n) = \max_{|q|=n} f_P(q)$$

Example. This definition does not correspond to the reality of CHR implementations. For example, a CHR-only machine can solve a crossword puzzle with time complexity linear in the number of words, which is impossible in implementations on a RAM machine. Given a query containing n words from a dictionary as `word/`k constraints (word length k), the following program returns all s solutions for the given crossword puzzle in time $n + s$.

A	B	C	D	E
F	■	G	■	H
I	J	K	■	L
■	M	N	O	P
Q	R	S	T	U

Crossword puzzle solver

```
word(A,B,C,D,E), word(I,J,K), word(M,N,O,P),
word(Q,R,S,T,U), word(A,F,I), word(J,M,R),
word(C,G,K,N,S), word(O,T), word(E,H,L,P,U)
    ==>
sol(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U).
```

3 Simulating Turing machines in CHR

Lemma 2. *Any Turing machine can be simulated on a CHR-only machine.*

Proof. Consider the following Turing machine simulator written in (pure) CHR:

Turing Machine Simulator

```
trans @ delta(S,G,Sp,Gp,Dir) \  current_state(S), head(Cell),
                                cell(Cell,G,Left,Right)
                             <=> current_state(Sp),
                                cell(Cell,Gp,Left,Right),
                                move(Dir,Cell,Left,Right).

move(l,Cell,L,_) <=> L \= null | head(L).
move(l,Cell,null,R), cell(Cell,G,null,R)
     <=> cell(Cell,G,L,R), cell(L,b,null,Cell), head(L).
move(r,Cell,_,R) <=> R \= null | head(R).
move(r,Cell,L,null), cell(Cell,G,_,_)
     <=> cell(Cell,G,L,R), cell(R,b,Cell,null), head(R).
```

The meaning of the different constraints in the simulator is:

- `delta/5` encodes the partial transition function in the obvious way. The first two arguments are the inputs and the last three the output of δ.
- `current_state/1` contains the current state.
- `head/1` contains the identifier of the cell under the head.

- `cell/4` represents a cell of the tape. The first argument is the unique identifier of the cell. The second is the symbol in the cell (`b` represents the blank symbol) and the last two are the identifiers of the neighboring cells.
- `move/4` represents a move of the head. The first argument is the direction of movement, `l` for left and `r` for right. The second argument is the identifier of the current cell and the last two are the identifiers of respectively the cell to the left and the right.

The special cell identifier `null` is used to refer to a not yet instantiated cell. The corresponding rules for `move/2` take care of extending the tape.

A simulation of the execution of a Turing machine M proceeds as follows. The the left-most and rightmost non-blank cells on the initial tape are encoded as `cell/4` constraints together with all the cells in between. The identifier of the cell to the left of the left-most non-blank is set to `null` and similarly for the cell to the right of the right-most non-blank cell. The transition function δ of M is encoded in multiple `delta/5` constraints. All these constraints are combined in the initial query together with the constraint `current_state(S0)` where `S0` is the initial state of M and the constraint `head(LeftMost)` where `LeftMost` is the identifier of the left-most non-blank cell.

The reader can easily verify that this CHR program can indeed simulate any Turing machine. Also, the mapping of Turing machine programs and tape inputs to the corresponding query for the simulator is obvious. Finally it is straightforward to extract the output of the simulated Turing machine from the output of the simulator: it suffices to look at the remaining `cell/4` constraints.

3.1 Time complexity

Theorem 1. *A Turing machine with time complexity $\mathcal{O}(T)$ can be simulated on the above CHR machine in time $\mathcal{O}(T)$.*

Proof. If T is the number of steps of the execution of the Turing Machine M, then the number of CHR transition steps in the simulator is bounded by $6T + 2E$. Every step of the Turing machine corresponds with one application of the `trans` rule and one application of either of the move rules. These two rules introduce at least 4 new constraints that need to be inserted. That accounts for the term $6T$. Finally, each time the tape is extended to the left or the right, two additional `cell/4` constraints are inserted by a move rule. E is the number of these extensions, with $E \leq T$.

Hence the time complexity of the simulation of M in CHR is $\mathcal{O}(T)$ where T is the time complexity of M. □

Simulator specialization. The simulator program may be specialized for a particular Turing Machine. Instead of the `trans` rule, the specialized version has one rule of the form:

```
    S, head(Cell), cell(Cell,G,Left,Right)
<=> Sp, cell(Gp,Left,Right), move(Dir,Cell,Left,Right).
```

for every `delta(S,G,Sp,Gp,Dir)` constraint. This specialization does not affect the time complexity in an essential way.

4 Simulating RAM machines in CHR

Lemma 3. *Any Peano-arithmetic RAM machine can be simulated on a CHR-only machine, and any standard-arithmetic RAM machine can be simulated on a minimal host-language CHR machine.*

4.1 Simulating the Peano-arithmetic RAM

The following CHR-only program simulates the Peano-arithmetic RAM machine:

Peano-arithmetic RAM simulator

```
prog(L,L1,inc,A) \ m(A,X), pc(L) <=> m(A,Z), s(Z,X), pc(L1).
prog(L,L1,dec,A) \ m(A,X), s(X,Y), pc(L) <=> m(A,Y), pc(L1).
prog(L,L1,clr,A) \ m(A,X), pc(L) <=> m(A,zero), pc(L1).

prog(L,L1,jump,A) \ pc(L) <=> pc(A).
prog(L,L1,cjump,A,L2), m(A,zero) \ pc(L) <=> pc(L2).
prog(L,L1,cjump,A,L2), m(A,X), s(X,_) \ pc(L) <=> pc(L1).

prog(L,L1,halt) \ pc(L) <=> true.
```

The query consists of an encoding of the RAM program as `prog/{3,4,5}` constraints; the first argument represents the label (or line number), the second argument is the label of the next program line, the third argument is an instruction mnemonic. The other arguments of the `prog/{4,5}` constraints contain the operands of the instruction. The memory content is represented as a chain of `s/2` constraints ending in `zero`, where `s/2` represents the successor function. The memory has to be initialized by adding to the query one `m/2` constraint for every register which is used in the program. Finally, the program counter must be set to the label of the first line of the program by adding an initial `pc/1` constraint. Note that this CHR program does not rely on numbers or arithmetic operations.

Time complexity. After the initialization, one rule is applied for every step of the Peano-Arithmetic RAM. Applying a rule causes at most three constraints to be inserted. Therefore, the number of CHR steps is bounded by four times the number of RAM steps. Hence, the time complexity of simulating a PA-RAM machine M in CHR(-only) is $\mathcal{O}(T)$ where M has time complexity T.

Initializing the n memory cells to values v_i $(0 \leq i \leq n)$ means inserting n `m/2` constraints and m `s/2` constraints, with $m = \sum_{i=1}^{n} v_i$.

Theorem 2. *A Peano-arithmetic RAM machine with time complexity $\mathcal{O}(T)$ can be simulated on the above CHR-only machine in time $\mathcal{O}(T)$, not counting memory initialization.*

4.2 Simulating the standard-arithmetic RAM

We can also simulate the more realistic standard-arithmetic RAM machine in CHR. However, if we want to do this without a complexity penalty, we need minimal host language operations to handle the integers. The following minimal host-language CHR program simulates the standard RAM machine. We used Prolog syntax here – but of course the five host-language statements can easily be modified to the syntax of other host-languages like Java.

Standard RAM simulator

```
prog(L,L1,const,B,A) \ m(A,_), pc(L) <=> m(A,B), pc(L1).
prog(L,L1,add,B,A), m(B,Y) \ m(A,X), pc(L) <=>
    Z is X+Y, m(A,Z), pc(L1).
prog(L,L1,sub,B,A), m(B,Y) \ m(A,X), pc(L) <=>
    Z is X-Y, m(A,Z), pc(L1).
prog(L,L1,mult,B,A), m(B,Y) \ m(A,X), pc(L) <=>
    Z is X*Y, m(A,Z), pc(L1).
prog(L,L1,div,B,A), m(B,Y) \ m(A,X), pc(L) <=>
    Z is X//Y, m(A,Z), pc(L1).

prog(L,L1,move,B,A), m(B,X) \ m(A,_), pc(L) <=> m(A,X), pc(L1).
prog(L,L1,i_move,B,A), m(B,C), m(C,X) \ m(A,_), pc(L) <=>
    m(A,X), pc(L1).
prog(L,L1,move_i,B,A), m(B,X), m(A,C) \ m(C,_), pc(L) <=>
    m(C,X), pc(L1).

prog(L,L1,jump,A) \ pc(L) <=> pc(A).
prog(L,L1,cjump,R,A), m(R,0) \ pc(L) <=> pc(A).
prog(L,L1,cjump,R,A), m(R,X) \ pc(L) <=> X =\= 0 | pc(L1).

prog(L,L1,halt) \ pc(L) <=> true.
```

Time complexity. The time complexity of this simulator is similar to the Peano-arithmetic RAM simulator. A minor difference is in the time for memory initialization, which does not depend on the size of the input numbers as in the Peano-arithmetic RAM simulator.

Theorem 3. *A standard-arithmetic RAM machine with time complexity $\mathcal{O}(T)$ can be simulated on the above minimal host-language CHR machine in time $\mathcal{O}(T)$, not counting memory initialization.*

5 Simulating CHR on RAM machines

For the simulation of CHR on RAM machines we can refer to the available CHR implementations in various host languages: Prolog [13, 9], HAL [10], Haskell [16]

and Java [2, 17]. As all these host languages can be simulated on RAM machines, the simulation of CHR on RAM machines follows.

Theorem 4. *Both CHR machines are Turing-complete.*

Proof. Because of Lemma 2, every Turing machine can be simulated on a CHR-only machine (which is also a minimal host-language CHR machine). Every CHR machine can be simulated on a Turing machine, because every CHR machine can be simulated on a RAM machine (we can use any CHR compiler), and because RAM machines can be simulated on a Turing machine (Lemma 1). □

Part of the non-determinism in the operational semantics of CHR is reduced in the above CHR systems, since they use specific instantiations of the abstract operational semantics. We consider the particular instantiation called the *refined operational semantics* [4], which is used in all major optimizing CHR systems.

5.1 Time complexity of simulating CHR on a RAM machine

The following general result holds for CHR-only programs and for CHR programs where every host language statement takes constant time.

Theorem 5. *A CHR program with time complexity $\mathcal{O}(f(n))$ can be simulated on a RAM machine with time complexity $\mathcal{O}(f(n)^m)$, where m is the maximal number of head constraints in a rule of the CHR program.*

Proof sketch. We will use the execution strategy of the refined operational semantics: after every **Introduce** transition, we try to apply, in textual order, the rules containing the introduced (or *active*) constraint in the head. The **Solve** transition takes constant time by assumption. The **Introduce** transition can also be performed in constant time, given a sufficiently efficient implementation of the constraint store – a linked list suffices. Trying an **Apply** transition can be done as follows: for every occurrence, find the partner constraints in the store and check the guard condition. In the worst case, this takes $\mathcal{O}(RS^{m-1})$ RAM cycles, where R is the number of occurrences of the active constraint and S is the number of constraints in the store. Because S is bounded by $\mathcal{O}(f(n))$, and R is bounded by a constant for a fixed CHR program, we get a worst-case time complexity of $\mathcal{O}(f(n)^{m-1})$ RAM cycles per CHR transition. Hence, the time for simulating $\mathcal{O}(f(n))$ CHR steps on a RAM machine is $\mathcal{O}(f(n)^m)$. □

Note that restricting the execution strategy to the refined operational semantics results in a tighter bound for the cost of trying an **Apply** transition. Lemma 4.1 in [6] gives a complexity of $\mathcal{O}(rS^m)$ where r is the number of CHR rules, which is weaker than our bound of $\mathcal{O}(RS^{m-1})$. The reason is that in the refined semantics not all possible combinations of constraints are tried: only the combinations containing the active constraint are considered.

5.2 RAM → CHR → RAM

We are now ready to state the most important result of this paper. It implies that every algorithm can be implemented in CHR with the best known time and space complexity.

Theorem 6. *For every RAM program P with time complexity $f(n)$, using $g(n)$ memory cells, a minimal host-language CHR program P' exists which computes output corresponding to the output of P. Furthermore, P' can be simulated on a RAM machine with time complexity $\mathcal{O}(f(n))$, using $\mathcal{O}(g(n))$ memory cells.*

Proof sketch. It suffices to show that for the RAM simulator described in Section 4.2, the host-language code generated by the CHR compiler can perform every RAM instruction in constant time, and the host-language code can be executed using $\mathcal{O}(g(n))$ memory. This is in fact the case for the K.U.Leuven CHR compiler [13], which uses intelligent scheduling of partner constraint lookups, and hash-table constraint stores for ground constraints. These features allow a constant-time lookup of the partner `prog/4` and `m/2` constraints since they are functionally dependent on the given `pc/1` constraint. Also, the insertion and deletion of `m/2` constraints in the store can be performed in constant time. Hence, the time complexity of the resulting RAM simulation of P' is linear in the number of rule applications (which is exactly $f(n)$).

Thanks to compile-time garbage collection, space can be reused every time a memory cell constraint `m(A,_)` is updated. Also, potential stack related additional space use can be reduced to a constant because of tail recursion optimization. The propagation history is only needed for propagation rules, which do not occur in the simulator program. Because of the above optimizations, the space usage will be linear in the number of `m/2` constraints (which is exactly $g(n)$). □

This means that if some algorithm can be implemented efficiently in some programming language, it can be implemented in CHR too, with the same time and space complexity. In other words, everything can be implemented in CHR with the best known complexity. Of course, in the worst case this can only be done by simulating the RAM version of an existing program. However, we are convinced that in practice much more elegant CHR programs can be constructed.

6 Simulating CHR on Turing machines

In the previous section, we showed that a CHR program with time complexity $\mathcal{O}(f(n))$ can be simulated on a RAM machine with time complexity $\mathcal{O}(f(n)^m)$. According to Lemma 1, RAM machines with time complexity T can be simulated on a Turing machine with time complexity $\mathcal{O}(T^8)$. Combining these upper bounds, we get the following (expected) result:

Theorem 7. *Any CHR machine with time complexity $\mathcal{O}(f(n))$ can be simulated on a Turing machine with time complexity $\mathcal{O}(f(n)^{8m})$, where m is the maximal number of head constraints in the rules of the CHR machine.*

Corollary 1. *Let* $\mathbf{P} = \mathbf{P}_{\mathrm{TM}}$ *be the class of problems that can be solved with a Turing machine in polynomial time, and* $\mathbf{P}_{\mathrm{CHR}}$ *the class of problems that can be solved with a CHR-only machine in polynomial time. We have* $\mathbf{P}_{\mathrm{TM}} = \mathbf{P}_{\mathrm{CHR}}$.

Proof. Follows from Theorem 1 and Theorem 7. □

7 Conclusion

We have generalized the optimal complexity result obtained for the union-find algorithm [15] to any algorithm which can be executed on a RAM machine. In other words, we showed that every algorithm can be implemented in CHR with the best-known time and space complexity.

We have shown that the CHR language is Turing-complete. We have introduced a new model of computation, the CHR machine. We have established upper bounds on the penalties for simulating Turing machines and RAM machines on a CHR machine, and vice versa, and found that all three models are polynomially related. This was done by providing and analyzing Turing machine and RAM machine simulators written in CHR.

Future work. An open problem is to formulate, if possible, the equivalent of the linear speedup theorem [8] for CHR machines. This would involve combining derivation steps by combining rules, improving the time complexity by a constant factor. It is unlikely that such a theorem would be useful in practice.

Another open question, related to database indexing, is a syntactic characterization of CHR programs that admit finding applicable rules in constant time.

There seems to be a similarity between non-deterministic Turing machines and CHR programs which allow multiple execution paths. For confluent programs, some paths will lead more quickly to a solution than others, and depending on the execution strategy, the program might have a different time complexity. Non-confluent programs are of course even more dependent on the specifics of the execution strategy. A non-deterministic Turing machine could be modelled as a non-confluent CHR program, where the non-deterministic choices correspond to execution strategy decisions.

Another almost unexplored topic is the study of what could be called a k-*parallel CHR machine* ($k \in \mathbb{N}_0 \cup \{\infty\}$) : a CHR machine able to perform at most k transitions simultaneously, as long as the multisets of removed constraints of every rule application are disjunct.

A variant of the RAM machine, the RASP machine (Random Access Stored Program), has its program in the working memory, allowing self-modifying programs. Both models are equivalent within a constant factor. The concept of self-modifying CHR programs needs to be investigated. Interestingly, self-modifying CHR programs are not polynomially related to the models discussed in this paper, since the number of head constraints in a rule would not remain constant for a given program.

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, Germany, September 2001.
3. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addisson-Wesley, 1975.
4. 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.
5. Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
6. Thom Frühwirth. As Time Goes By: Automatic Complexity Analysis of Concurrent Rule Programs. In *Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning*, Toulouse, France, April 2002.
7. Harald Ganzinger and David McAllester. A new meta-complexity theorem for bottom-up logic programs. In *Proceedings of the First International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 514–528, Siena, Italy, June 2001. Springer Verlag.
8. Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Trans. American Mathematical Society*, 117:285–306, May 1965.
9. Christian Holzbaur and Thom Frühwirth. Compiling constraint handling rules into Prolog with attributed variables. In Gopalan Nadathur, editor, *Proceedings of the Intl. Conference on Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 117–133. Springer Verlag, 1999.
10. 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 CHR*, 5(4+5), 2005.
11. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Longman, 2001.
12. John E. Savage. *Models of Computation*. Addison-Wesley, 1998.
13. 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*, Ulm, Germany, May 2004.
14. Tom Schrijvers, Bart Demoen, Gregory Duck, Peter Stuckey, and Thom Frühwirth. Automatic Implication Checking for CHR Constraint Solvers. Report CW 402, K.U.Leuven, Department of Computer Science, Leuven, Belgium, January 2005.
15. Tom Schrijvers and Thom Frühwirth. Optimal Union-Find in Constraint Handling Rules. *Theory and Practice of Logic Programming*, 2005. To appear.
16. Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. *ACM Transactions on Programming Languages and Systems*, 2005. To appear.
17. Armin Wolf. Adaptive Constraint Handling with CHR in Java. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science 2239. Springer Verlag, 2001.

A Type System for CHR

Emmanuel Coquery[1,2] and François Fages[1]

[1] INRIA Rocquencourt, Projet Contraintes,
BP 105, F-78153 Le Chesnay, France
[2] Conservatoire National des Arts et Métiers,
292 rue Saint Martin, 75141 Paris Cedex 03, France

Abstract. We propose a generic type system for the *Constraint Handling Rules* (CHR), a rewriting rule language for implementing constraint solvers. CHR being a high-level extension of a host language, such as Prolog or Java, this type system is parameterized by the type system of the host language. We show the consistency of the type system for CHR w.r.t. its operational semantics. We also study the case when the host language is a constraint logic programming language, typed with the prescriptive type system we developed in our previous work. In particular, we show the consistency of the resulting type system w.r.t. the extended execution model CLP+CHR. This system is implemented through an extension of our type checker TCLP for constraint logic languages. We report on experimental results about the type-checking of 12 CHR solvers and programs, including TCLP itself.

1 Introduction

The language of *Constraint Handling Rules* (CHR) of T. Frühwirth [1] is a successful rule-based language for implementing constraint solvers in a wide variety of domains. It is an extension of a host language, such as Prolog [2], Java [3] or Haskell [4], allowing the introduction of new constraints in a declarative way. CHR is used to handle user-defined constraints while the host language deals with other computations using *native* constraints. CHR is a commited-choice language of guarded rules that rewrite constraints into simpler ones until they are in solved forms. One peculiarity of CHR is that it allows multiple heads in rules.

Typed languages have numerous advantages from the point of view of program development, such as the static detection of programming errors or program composition errors, and the documentation of the code by types. CHR has already been used for the typing of programming languages, either for solving subtyping constraints [5, 6] or for handling overloading in functional languages [7] and constraint logic languages [8, 6]. There were however only few works the the typing of CHR itself. In [4], Chin, Sulzmann and Wang propose a monomorphic type system for the embedding of CHR into Haskell.

In this article, we propose a generic type system for CHR inspired by the TCLP type system for constraint logic programs [9]. CHR being an extension

of a host language, this system is parameterized by the type system of the host language. We will make three assumptions on the type system of the host language:

- Typing judgments of the form $\Gamma \vdash t : \tau$ are considered, where τ is a type associated to the term t in a typing environment Γ. Moreover *well-typed* constraints in a typing environment Γ are defined by a derivation system for typing judgments.
- The constraint $t_1 = t_2$ is well-typed in the environment Γ if there exists a type τ such as $\Gamma \vdash t_1 : \tau$ and $\Gamma \vdash t_2 : \tau$.
- If a conjunct c of native constraints is well-typed in an environment Γ and is equivalent to a conjunct d, then d is also well-typed in Γ.

Using these assumptions, we show the consistency of the type system for CHR w.r.t. its operational semantics. This is expressed by a subject reduction theorem which establishes that if a program is well-typed then all the derived goals from a well-typed goal are well-typed.

We also study the instantiation of this type system with the TCLP type system for constraint logic programs [9]. We show a subject reduction theorem for the CLP+CHR execution model [1] in which it is possible to extend the definition of constraints by clauses. This result is interesting because constraint logic programming is a natural framework for using constraint solvers. A type system for CLP+CHR allows us to type-check a solver together with the program that uses it, as well as complex CHR solvers written as a combination of clauses and rules, where CHR rules use CLP predicates and CLP clauses post CHR constraints.

The rest of the paper is organized as follows. Section 2 recalls the syntax and operational semantics of CHR, including the CLP+CHR execution model. Section 3 presents the type system and section 4 presents its instantiation with the type system for CLP. Section 5 presents some experimental results on the typing of some CHR solvers, using the implementation of the system in TCLP [10]. Finally, we conclude in section 6.

2 Preliminaries on CHR

Here, we recall the syntax and semantics of CHR, as given in [1]. We distinguish the user-defined *CHR constraints* from the *native constraints* of the host language, which represent auxiliary computations that take place during the application of a CHR rule. We assume that native constraints are handled by a predefined solver of the host language. We also assume that native constraints include the equality constraint $= /2$ and the constraint *true*. The terms of the host language are noted $s, t, \ldots$ We note $\mathcal{X}$ the domain of native constraints, and $\mathcal{CT}$ its (possibly incomplete) first-order logic theory.

2.1 Syntax

Definition 1. *A CHR rule is either:*

– *a* simplification *rule of the form:*
  $H_1, \ldots, H_i$ `<=>` $G_1, \ldots, G_j \mid B_1, \ldots, B_k$
– *a* propagation *rule of the form:*
  $H_1, \ldots, H_i$ `==>` $G_1, \ldots, G_j \mid B_1, \ldots, B_k$
– *or a* simpagation *rule of the form:*
  $H_1, \ldots, H_l \setminus H_{l+1}, \ldots, H_i$ `<=>` $G_1, \ldots, G_j \mid B_1, \ldots, B_k$

with $i > 0$, $j \geq 0$, $k \geq 0$, $l > 0$ and $H_1, \ldots, H_i$ is a nonempty sequence of CHR constraints, the guard $G_1, \ldots, G_j$ being a sequence of native constraints and the body $B_1, \ldots, B_k$ being a sequence of CHR and native constraints.

A CHR program is a finite sequence of CHR rules.

The constraint *true* is used to represent empty sequences. The empty guard can be omitted, together with the | symbol. The notation $name@R$ gives a name to a CHR rule R.

Informally, a simplification rule replaces the constraints of the head by the constraints of the body. A propagation rule adds the constraints of the body while keeping the constraints of the head in the store. A simpagation rule is a mix of the two preceding kind of rules: the constraints $H_{l+1}, \ldots, H_i$ are replaced by the body, while the constraints $H_1, \ldots, H_l$ are kept.

For the sake of simplicity, and because the distinction of propagation and simpagation rules are not needed for typing purposes, we will consider that a propagation or a simpagation rule of the form

$$H_1, \ldots, H_l \setminus H_{l+1}, \ldots, H_i \texttt{ <=> } G_1, \ldots, G_j \mid B_1, \ldots, B_k$$

is just an abbreviation for the simplification rule

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

Example 1. The following CHR program, taken from [1], defines a solver for a general ordering constraint `=<`.

```
reflexivity  @ X=<Y <=> X=Y | true.
antisymetry  @ X=<Y , Y=<X <=> X=Y.
transitivity @ X=<Y , Y=<Z ==> X=<Z.
identity     @ X=<Y \ X=<Y <=> true.
```

The rule `reflexivity` eliminates the `=<` constraints when its two arguments are equal. Rule `antisymmetry` simplifies a double inequality into an equality. The rule `transitivity` adds constraints corresponding to the transitive closure of `=<`. Finally, `identity` eliminates redundant `=<` constraints.

2.2 Operational Semantics

The operational semantics of CHR is expressed by a transition system, noted $\longmapsto$, over states which are triples $\langle F, E, D \rangle$, where F is a goal, that is a multiset

of native and CHR constraints, E is a CHR constraint store and D is a native constraint store. A state is thus a conjunction of CHR and native constraints.

In the following definition, the equality is extended to constraints by morphism, that is $c(t_1, \ldots, t_n) = c(t'_1, \ldots, t'_n)$ if $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n$. The conjunction notation $\wedge$ is used to express the matching of a constraint in a multiset, The equality is also extended to conjunctions of constraints: $H_1 \wedge \ldots \wedge H_n = H'_1 \wedge \ldots \wedge H'_n$ if $H_1 = H'_1 \wedge \ldots \wedge H_n = H'_n$.

Definition 2. *Let P be a CHR program. The transition relation $\longmapsto$ is given by the following rules, where the variables appearing in triples stand for conjunctions of constraints and $\bar{x}$ represents the set of variables appearing in the head H.*

Solve
$\langle C \wedge F, E, D\rangle \longmapsto \langle F, E, D'\rangle$
if C is a native constraint and $\mathcal{CT} \models (C \wedge D) \Leftrightarrow D'$.

Introduce
$\langle H \wedge F, E, D\rangle \longmapsto \langle F, H \wedge E, D\rangle$
if H is a CHR constraint.

Simplify
$\langle F, H' \wedge E, D\rangle \longmapsto \langle B \wedge F, E, H = H' \wedge D\rangle$
if (H `<=>` $G \mid B$) is in P renamed with fresh variables,
and $\mathcal{CT} \models D \Rightarrow \exists \bar{x}(H = H' \wedge G)$.

Propagate
$\langle F, H' \wedge E, D\rangle \longmapsto \langle B \wedge F, H' \wedge E, H = H' \wedge D\rangle$
if (H `==>` $G \mid B$) is in P renamed with fresh variables,
and $\mathcal{CT} \models D \Rightarrow \exists \bar{x}(H = H' \wedge G)$.

The **Solve** transition corresponds to a transition of the native constraint solver. The **Introduce** transition simply transfers a CHR constraint from the goal to the CHR constraint store The **Simplify** transition correspond to the application of CHR simplification. The **Propagate** transition is indicated for the sake of clarity, although it is treated as an abbreviation for a simplification rule in the rest of the paper. The condition for applying these rules is that the head of the rule can be instantiated such that the guard and the matching condition of the head are implied by the current native constraint store. The body of the rule is then added to the current goal and, when applying a **Simplify** transition, the constraints matching the head are removed from the constraint store.

Definition 3. *An* initial state *consists in a goal F and two empty constraint stores: $\langle F, true, true\rangle$. A* final state *is either of the form $\langle F, E, false\rangle$ (*failure*), or of the form $\langle true, E, D\rangle$ where D is satisfiable (*success*).*

The following example illustrates the execution of a CHR program.

Example 2. Let us consider the solver given in example 1 together with the initial state $\langle$`X=<Y` $\wedge$ `Y=<Z` $\wedge$ `Z=<X`, *true*, *true*$\rangle$. One possible execution is:

$\langle$ `Z=<X, X=<Y` $\wedge$ `Y=<Z`, *true*$\rangle$ (**Introduce** $\times 2$)
$\langle$ `X=<Z` $\wedge$ `Z=<X, X=<Y` $\wedge$ `Y=<Z`, *true*$\rangle$ (**Propagate** `transitivity`)
$\langle$ *true*, `X=<Z` $\wedge$ `Z=<X` $\wedge$ `X=<Y` $\wedge$ `Y=<Z`, *true*$\rangle$ (**Introduce** $\times 2$)
$\langle$ `X=Z, X=<Y` $\wedge$ `Y=<Z`, *true*$\rangle$ (**Simplify** `antisymmetry`)
$\langle$ *true*, `X=<Y` $\wedge$ `Y=<Z, X=Z`$\rangle$ (**Solve**)
$\langle$ `X=Y`, *true*, `X=Z`$\rangle$ (**Simplify** `antisymmetry`)
$\langle$ *true*, *true*, `X=Y` $\wedge$ `X=Z`$\rangle$ (**Solve**)

One can remark that in this operational semantics, once a propagation rule can be applied, it can be applied infinitely often, which leads to a trivial case of non termination. In the preceding example, one could have applied the `transitivity` rule instead of the `antisymmetry` rule, thus reintroducing the constraint `X=<Z` that was eliminated at the fourth step. In [11], Abdennadher gives refined operational semantics that are more faithful to the actual implementation of CHR. In particular the previous behavior is avoided by restricting the application of a rule only once on the same constraints. The subject reduction theorems given in the following sections express that given a well-typed program, a transition occurring from a well-typed state leads to a well-typed state. It is worth noting that they thus hold also in these more realistic semantics.

2.3 CLP+CHR

When the host language is a constraint logic programming language of the class $CLP(\mathcal{X})$ [12], it is possible to tightly integrate CHR to the host language. To this end, Frühwirth [1] proposed to extend CHR with the construct *label_with* used to define CHR constraints by CLP clauses. We recall here the syntax and operational semantics of this extension. We note $\mathcal{S}_F$ (resp. $\mathcal{S}_P$) the set of function (resp. predicate) symbols, given with their arity, and $\mathcal{V}$ the set of variables. An *atom* is either a native constraint, a CHR constraint or of the form $p(t_1, \ldots, t_n)$, where p/n is a program predicate symbol. The declaration *label_with* $c(t_1, \ldots, t_n)$ *if* $G_1, \ldots, G_j$ expresses that $G_1, \ldots, G_j$ is a guard for the clauses of the CHR constraint c/n.

Definition 4. *A* labeling declaration *for a CHR constraint H is an expression of the form:*

label_with H *if* $G_1, \ldots, G_j$

where $G_1 \ldots, G_j$ is a conjunction of native constraints.
Clauses are of the form:

H `:-` $B_1, \ldots, B_n$

where H an atom corresponding either to a predicate or to a CHR constraint but not to a native constraint, and $B_1, \ldots, B_k$ is a sequence of atoms.

Definition 5. *The relation transition between CHR states is extended by the two following rules:*

Unfold

$\langle H' \wedge F, E, D\rangle \longmapsto \langle B \wedge F, E, H = H' \wedge D\rangle$

if $(H$ *:-* $B)$ *is in* P *renamed with fresh variables,*
and H *is not a CHR constraint.*

Label

$\langle F, H' \wedge E, D\rangle \longmapsto \langle B \wedge F, E, H = H' \wedge D\rangle$

if $(H$ *:-* $B)$ *and (label_with* H'' *if* $G)$ *are in* P *renamed with fresh variables,*
and $\mathcal{CT} \models D \Rightarrow \exists \bar{x}(H' = H'' \wedge G)$

The **Unfold** transition is close to the CSLD resolution rule [12]. The difference is that, under CSLD resolution, the constraints in the body of the resolving clause are added to the native constraint store and the resulting store, i.e. $H = H' \wedge D \wedge C$, must be satisfiable, which is not demanded here. The CLP clauses for CHR constraints can only be used in a **Label** transition, requiring that the guards declared using *label_with* are implied by the current native constraint store.

3 Type System

3.1 Assumptions about the type system of the host language

Since CHR is an extension of a host language, the type system we propose is parameterized by the type system, noted $\vdash_N$, of the host language. We will make the following assumptions on $\vdash_N$.

We suppose that $\vdash_N$ is based on a type algebra, the set of types being noted $\mathcal{T}$. Types are noted using the letter τ. Typing environments, noted Γ, associate types to program variables. Given an expression t and a typing environment Γ, $\vdash_N$ is used to deduce typing judgments of the form $\Gamma \vdash_N t : \tau$. Similarly, $\vdash_N$ is used to deduce *well-typed* constraints in a typing environment Γ, a conjunction $C_1 \wedge \ldots \wedge C_n$ of native constraints being well-typed in Γ if for each $i \in \{1, \ldots, n\}$, C_i is well-typed in Γ. We note $\Gamma \vdash_N C$ *Atom*, the fact that C is well-typed in the typing environment Γ. We also assume that the equality constraint $s = t$ between s and t is well-typed in Γ if there exists a type τ such that $\Gamma \vdash_N s : \tau$ and $\Gamma \vdash_N t : \tau$.

We assume that the union of type environments over disjoint sets of variables can be formed with an operation noted $\uplus$ such that if $\Gamma \vdash_N t : \tau$ then $\Gamma \uplus \Gamma' \vdash_N t : \tau$ for any typing environment Γ' disjoint from Γ. We also assume that if a conjunction of native constraints C is well-typed in a typing environment Γ and $\mathcal{CT} \models C \Leftrightarrow D$, then there exists a typing environment Γ', such that the conjunction of constraints D is well-typed in $\Gamma \uplus \Gamma'$.

3.2 Type System for CHR

The type system we propose for CHR defines a notion of well-typedness for CHR rules. To each CHR constraint symbol c/n is associated a set of types $types(c/n)$, each type being of the form $\tau_1 \times \ldots \times \tau_n$. This set of types is assumed

to be fixed, for example using some declarations provided by the programmer. This framework allows the use of parametric polymorphism [13]. A parametric type scheme $\forall\alpha_1 \ldots \alpha_k.\ \tau_1 \times \ldots \times \tau_n$ is represented by the set of all its possible instantiations. For example, declaring that $types(\texttt{append/3}) = \{list(\tau) \times list(\tau) \times list(\tau) \mid \tau \in \mathcal{T}\}$ allows one to give the type $\forall\alpha.\ list(\alpha) \times list(\alpha) \times list(\alpha)$ to the constraint `append/3`.

$$(Native)\quad \frac{\Gamma \vdash_N C\ Atom}{\Gamma \vdash C\ Atom} \qquad \text{if } C \text{ is a native constraint}$$

$$(CHR\ Atom)\quad \frac{\Gamma \vdash_N t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash_N t_n : \tau_n}{\Gamma \vdash c(t_1, \ldots, t_n)\ Atom} \qquad \begin{array}{r}\text{if } c/n \text{ a CHR constraint}\\ \text{and if } \tau_1 \times \ldots \times \tau_n \in types(c/n)\end{array}$$

$$(Goal)\quad \frac{\Gamma \vdash B_1\ Atom \quad \ldots \quad \Gamma \vdash B_n\ Atom}{\Gamma \vdash B_1, \ldots, B_n\ Goal}$$

$$(CHR\ Head)\quad \frac{\Gamma \vdash_N t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash_N t_n : \tau_n}{\Gamma \vdash c(t_1, \ldots, t_n)\ Head_{\tau_1 \times \ldots \times \tau_n}} \qquad \begin{array}{r}\text{if } c/n \text{ a CHR constraint}\\ \text{and if } \tau_1 \times \ldots \times \tau_n \in types(c/n)\end{array}$$

$$(MultiHead)\quad \frac{\Gamma \vdash H_1\ Head_{\sigma_1} \qquad \ldots \qquad \Gamma \vdash H_i\ Head_{\sigma_i}}{\Gamma \vdash H_1, \ldots, H_i\ MHead_{\sigma_1, \ldots, \sigma_i}}$$

$$(Simpl\ CHR)\quad \frac{\begin{array}{ll}\forall(\sigma_1, \ldots, \sigma_n) \in S_1 \times \ldots \times S_n & \Gamma_{\sigma_1,\ldots,\sigma_n} \vdash H_1, \ldots, H_n\ MHead_{\sigma_1,\ldots,\sigma_n}\\ & \Gamma_{\sigma_1,\ldots,\sigma_n} \vdash G_1, \ldots, G_r\ Goal\\ & \Gamma_{\sigma_1,\ldots,\sigma_n} \vdash B_1, \ldots, B_q\ Goal\end{array}}{\vdash H_1, \ldots, H_n\ \texttt{<=>}\ G_1, \ldots, G_r \mid B_1, \ldots, B_q\ Rule}$$

where for all $i \in \{1, \ldots, n\}$, $H_i = c_i(t_1^i, \ldots, t_{m_i}^i)$ and $S_i = types(c_i/m_i)$

Table 1. Type system for CHR

The rules of the type system for CHR are given in table 1, where σ's represent types of CHR constraints and S's represent sets of such types.

The typing rules resemble to the rules of Chin, Sulzmann and Wang [4], with the addition of the possibility to have more than one type for a CHR constraint and the abstraction w.r.t. the host type system.

A CHR constraint H is *well-typed* in Γ if the judgment $\Gamma \vdash H\ Atom$ can be derived from the typing rule. Terms or expressions appearing as arguments of the constraints are typed using the type system $\vdash_N$ for the host language.

The rules (*CHR Head*) and (*MultiHead*), differ from (*CHR Atom*) and (*Goal*) in that they add an annotation for keeping track of the type used for typing each head. The rule (*Simpl CHR*) requires, for each combination $\sigma_1, \ldots, \sigma_n$ of the types of the different occurrences of the CHR constraints of the head of the CHR rule, that the head, the guard and the body of the CHR rule are well-typed in some typing environment $\Gamma_{\sigma_1,\ldots,\sigma_n}$. As shown in section 4.3, in the case of parametric polymorphism, this can be ensured by renaming the type scheme

of each occurrence of CHR constraints in the head with distinct variables, which can be seen as applying the principle of *definitional genericity* [14] to the typing of CHR constraints. In the context of logic programming, this principle establishes that the type of the head of a clause must be equivalent to, up to renaming but not an instance of, the declared type of the predicate.

The consistency of the type system w.r.t. the operational semantics of CHR is given by the following subject reduction theorem, which expresses that the well-typedness of goals is preserved by transitions:

Theorem 1. *Let P be a well-typed CHR program. Let $\langle F, E, D\rangle$ and $\langle F', E', D'\rangle$ be two states such that $\langle F, E, D\rangle \longmapsto \langle F', E', D'\rangle$. If there exists a typing environment Γ such that $\Gamma \vdash F, E, D$ Goal, then there exists a typing environment Γ' such that $\Gamma' \vdash F', E', D'$ Goal. Moreover, if the transition rule contains a guard G then $\Gamma' \vdash G$ Goal.*

The following example show the necessity of considering all possible combinations of types when typing a CHR rule with multiple heads.

Example 3. Let us assume that the constraint `=<` has the type scheme $\forall\alpha.\alpha \times \alpha$. Let us consider the polymorphic type $list(\alpha)$ for lists and the types int and $string$. We assume that the empty list `[]` has type $\forall\alpha.\ list(\alpha)$, that the list constructor has type $\forall\alpha.\ \alpha \times list(\alpha) \to \alpha$, and that $list(int)$ and $list(string)$ are incompatible. Then the `transitivity` rule is not well-typed:

```
X=<Y, Y=<Z  ==> X=<Z
```

For example, one might consider the type $list(int) \times list(int)$ for the first occurrence of `=<`$/2$ and the type $list(string) \times list(string)$ for the second one, in which case the head is not well-typed because `Y` can not have both types $list(int)$ and $list(string)$.

The rule can produce an ill-typed state from a well-typed one. The state $\langle true, \texttt{["a"] =< []} \wedge \texttt{[] =< [1]}, true\rangle$ is well-typed. However the rule would add $\texttt{X = ["a"]} \wedge \texttt{Z = [1]} \wedge \texttt{X =< Z}$ to the current goal. This subgoal is not well-typed because, when typing `X =< Z`, the type chosen for `=<`$/2$ must be compatible both with $list(string)$ and $list(int)$.

4 Integration with CLP

In this section we are interested in the particular case where the host language is a constraint logic language, typed using the prescriptive type system TCLP [9]. This system combines parametric polymorphism, subtyping and overloading to obtain the flexibility that is needed for typing CLP programs that are originally untyped. In particular, subtyping is used for typing the simultaneous use of different constraint domains: for instance, the relation $boolean \leq int$ allows one to see booleans as integers, and thus to type check constraints combining boolean variables with integer variables (such as in a sum of boolean variables). Subtyping is also used for the typing of programs using meta-programming techniques: the

relation $list(\alpha) \leq term$ allows one to see homogeneous lists as terms and to apply decomposition predicates to them, such as `functor/3`, `arg/3` or `=../2`.

In [9], the type system of TCLP is proved consistent w.r.t. the CSLD execution model [12], which is an abstract model of execution proceeding by constraint accumulation. In particular, the transformations that can be made by the constraint solver are not considered. In the following, we assume that the solver for native constraints only performs simplifications that preserve well-typedness, according to the assumptions of section 3.1. This can be obtained either by using a typed execution model, as proposed in [9], or, in the case of the equality constraint, by using modes to fix the dataflow [15].

First, we present the type algebra used in the system, then we recall the typing rules for CLP, together with a typing rule for the labeling declaration *label_with*. The resulting system is proven consistent w.r.t. the CLP+CHR execution model.

4.1 Type Structure

We consider a partial order $(\mathcal{K}, \leq_{\mathcal{K}})$ of *type constructors*, given with their arity. The set $\mathcal{T}$ of types is the set of finite or infinite types built on $\mathcal{K}$.

Subtyping Relation The use of subtyping for meta-programming purposes requires to consider relations like $list(\alpha) \leq term$. This form of non-structural non-homogeneous subtyping links different constructors of different arities. Such subtyping relations require to express the correspondence between the different arguments of type constructors. For example, by writing $k_1(\alpha, \beta) \leq k_2(\beta)$, we specify that types built with k_1 are subtypes of those built with k_2, provided that the second argument of k_1 is a subtype of the argument of k_2, the first argument of k_1 being forgotten in the subtyping relation. One way to express the correspondence is to use a formalism of labels, as proposed by Pottier [16]. In this formalism, a label is associated to each argument of type constructors, the correspondence being expressed by the fact that two arguments of type constructors have the same label. The subtyping order $\leq$ is built from the order $\leq_{\mathcal{K}}$ on type constructors and from the labels. A formal description of the type structure is given in [5], where the structures of types and type constructors are quasi-lattices, i.e. partial orders in which two elements have a least upper (resp. greatest lower) bound if and only if they have an upper (resp. lower) bound.

Subtyping constraints Let $\mathcal{W}$ be a set of *type variables*, or *parameters*, noted $\alpha, \beta, \ldots$. We note $\mathcal{T}_{\mathcal{W}}$ the set of types built on $\mathcal{K} \cup \mathcal{W}$.

Definition 6. *A* subtyping constraint *is of the form* $\tau_1 \leq \tau_2$*, where* $\tau_1, \tau_2 \in \mathcal{T}_{\mathcal{W}}$ *are finite types. A substitution* $\rho : \mathcal{W} \to \mathcal{T}$ satisfies *the constraint* $\tau_1 \leq \tau_2$*, noted* $\rho \models \tau_1 \leq \tau_2$*, if* $\rho(\tau_1) \leq \rho(\tau_2)$*. The subtyping constraint* $\tau_1 \leq \tau_2$ *is* satisfiable *if there exists a substitution* ρ *such that* $\rho \models \tau_1 \leq \tau_2$*.*

In [5], sufficient conditions on $(\mathcal{K}, \leq_{\mathcal{K}})$ are given for the decidability of the satisfiability of subtyping constraints in quasi-lattices, this problem is shown to NP-complete, and a practical algorithm (used in section 5) is given for computing explicit solutions.

4.2 Type system for CLP+CHR

In order to support the overloading of CLP function and predicate symbols, we assume that a set $types(f/n)$ of type schemes of the form $\forall\bar{\alpha}.\tau_1 \times \ldots \times \tau_n \to \tau$ is associated to each function symbol f/n (resp. predicate symbol p/n), where $\bar{\alpha}$ is the set of parameters occurring in types $\tau_1, \ldots, \tau_n, \tau$. These sets of types are supposed to be fixed, for example using declarations provided by the programmer. We also assume that the type of the constraint $= /2$ is the type scheme $\forall\alpha.\alpha \times \alpha$. For the sake of simplicity, the quantification $\forall\bar{\alpha}$ will be omitted in type schemes, each occurrence of a type scheme being renamed with fresh parameters.

A typing environment is a partial mapping $\Gamma : \mathcal{V} \mapsto \mathcal{T}_{\mathcal{W}}$, also noted $\{X_1 : \tau_1, \ldots, X_n : \tau_n\}$. The operation $\uplus$ on typing environments is defined as disjoint union, that is $(\Gamma_1 \uplus \Gamma_2)(X) = \Gamma_1(X)$ if $X \in dom(\Gamma_1)$, $(\Gamma_1 \uplus \Gamma_2)(X) = \Gamma_2(X)$ if $X \in dom(\Gamma_2)$, and $(\Gamma_1 \uplus \Gamma_2)(X)$ is undefined otherwise.

$$(Var)\quad \frac{X : \tau \in \Gamma}{\Gamma \vdash X : \tau} \qquad (Sub)\quad \frac{\Gamma \vdash t : \tau \quad \tau \leq \tau'}{\Gamma \vdash t : \tau'}$$

$$(Func)\quad \frac{\Gamma \vdash t_1 : \tau_1\rho \quad \ldots \quad \Gamma \vdash t_n : \tau_n\rho}{\Gamma \vdash f(t_1, \ldots, t_n) : \tau\rho} \quad \begin{array}{r}\rho \text{ is a type substitution} \\ \tau_1 \times \ldots \tau_n \to \tau \in types(f/n)\end{array}$$

$$(Atom)\quad \frac{\Gamma \vdash t_1 : \tau_1\rho \quad \ldots \quad \Gamma \vdash t_n : \tau_n\rho}{\Gamma \vdash p(t_1, \ldots, t_n)\ Atom} \quad \begin{array}{r}\rho \text{ is a type substitution} \\ \tau_1 \times \ldots \tau_n \in types(p/n)\end{array}$$

$$(Head)\quad \frac{\Gamma \vdash t_1 : \tau_1\rho \quad \ldots \quad \Gamma \vdash t_n : \tau_n\rho}{\Gamma \vdash p(t_1, \ldots, t_n)\ Head_{\tau_1 \times \ldots \times \tau_n}} \quad \begin{array}{r}\rho \text{ is a type renaming} \\ \tau_1 \times \ldots \tau_n \in types(p/n)\end{array}$$

$$(Clause)\quad \frac{\forall\sigma \in types(p/n) \quad \begin{array}{c}\Gamma_\sigma \vdash p(t_1, \ldots, t_n)\ Head_\sigma \\ \Gamma_\sigma \vdash B_1\ Atom \quad \ldots \quad \Gamma_\sigma \vdash B_k\ Atom\end{array}}{\vdash p(t_1, \ldots, t_n)\ \texttt{:-}\ B_1, \ldots, B_k\ Clause}$$

$$(Label\ with)\quad \frac{\Gamma \vdash H\ Atom \quad \Gamma \vdash G\ Goal}{\vdash label_with\ H\ if\ G\ Label_with}$$

Table 2. Type system for CLP and *label_with*

Table 2 gives the typing rules for CLP, together with the typing rule for the declaration *label_with*. The typing rules for CLP resemble the rules of Mycroft and O'Keefe [13] with the addition of subtyping and overloading. A predicate

call $p(t_1, \ldots, t_n)$ (resp. a native constraint) is *well-typed* in a typing environment Γ if $\Gamma \vdash p(t_1, \ldots, t_n)$ *Atom* can be derived from the rules. A clause H :- B is *well-typed* if $\vdash H$:- B *Clause* can be derived from the rules. A labeling declaration *label_with H if G* is well-typed if $\vdash$ *label_with H if G Label_with* can be derived. The (*Sub*) rule gives the semantics of subtyping by expressing that if a term t has type τ, then it has all types that are greater than τ.

The set of rules of tables 1 and 2 define the type system for CLP+CHR. A CLP+CHR program is well-typed if all its CHR rules, all its clauses and all its labeling declarations are well-typed.

The distinction between rules (*Atom*) and (*Head*) expresses the principle of *definitional genericity* [14], which establishes that the type of the head of a clause must be equivalent to, up to renaming but not an instance of, the declared type of the predicate. The rule (*Clause*) imposes that a clause must be well-typed for all possible types of the defined predicate, in a typing environment Γ_σ that depends on the considered type σ. This can be seen as a condition similar to definitional genericity for overloading. These two conditions are useful for the following subject reduction theorem which expresses the consistency of the type system with reference to the operational semantics of CLP+CHR.

Theorem 2. *Let us consider a well-typed CHR+CLP program. Let* $\langle F, E, D\rangle$ *and* $\langle F', E', D'\rangle$ *be two states and* Γ *be a typing environment such that* $\Gamma \vdash F, E, D$ *Goal. If* $\langle F, E, D\rangle \longmapsto \langle F', E', D'\rangle$*, then there exists a typing environment* Γ' *such that* $\Gamma' \vdash F', E', D'$ *Goal. Moreover, if the transition rules contains a guard* G *then* $\Gamma' \vdash G$ *Goal.*

4.3 Typing of polymorphic CHR constraints

In this section, we show how to type check CHR constraints in the presence of parametric polymorphism. The set $types(c/n)$ of the types of the constraint c/n is restricted to a finite set of type schemes of the form $\forall\bar{\alpha}.\ \tau_1 \times \ldots \times \tau_n$. This does not mean that $types(c/n)$ itself is finite, as a type scheme represent an infinite set of types. More precisely, we assume a finite set $types_p(c/n)$ of type schemes and define $types(c/n) = \{\tau_1\rho \times \ldots \times \tau_n\rho \mid \forall\bar{\alpha}.\ \tau_1 \times \ldots \times \tau_n \in types_p(c/n)$ and $\rho : \bar{\alpha} \to \mathcal{T}\}$.

A type system that deals directly with parametric polymorphism can be obtained by replacing rules (*CHR Atom*), (*CHR Head*) and (*MultiHead*) by their counterparts given in table 3 and by replacing $types(c_i/m_i)$ is replaced by $types_p(c_i/m_i)$ in rule (*Simpl CHR*). The resulting type system is noted $\vdash_p$. The following proposition expresses the equivalence of the two type systems:

Proposition 1. *A CHR rule (resp. goal) is well-typed in* $\vdash$ *if and only if it is well-typed in* $\vdash_p$*.*

5 Experimental Results

The type system for CLP+CHR has been implemented as an extension of the TCLP software [10], which is a type checker for constraint logic programming.

$$(CHR\ Atom') \quad \frac{\Gamma \vdash t_1 : \tau_1\rho \qquad \ldots \qquad \Gamma \vdash t_n : \tau_n\rho}{\Gamma \vdash c(\tau_1, \ldots \tau_n)\ Atom} \quad \begin{array}{r}\rho \text{ is a type substitution} \\ \forall\bar{\alpha}.\ \tau_1 \times \ldots \times \tau_n \in types_p(c/n)\end{array}$$

$$(CHR\ Head_p) \quad \frac{\Gamma \vdash t_1 : \tau_1\rho \qquad \ldots \qquad \Gamma \vdash t_n : \tau_n\rho}{\Gamma \vdash c(\tau_1, \ldots \tau_n)\ Head_{\tau_1 \times \ldots \times \tau_n, \rho}} \quad \begin{array}{r}\rho \text{ is a type renaming} \\ \forall\bar{\alpha}.\ \tau_1 \times \ldots \times \tau_n \in types_p(c/n)\end{array}$$

$$(MultiHead_p) \quad \frac{\begin{array}{c}\forall i \in \{1, \ldots, n\}\ \Gamma \vdash H_i\ Head_{\sigma_i, \rho_i} \\ \forall\ 1 \leq i < j \leq n\ codom(\rho_i) \cap codom(\rho_j) = \emptyset\end{array}}{\Gamma \vdash H_1, \ldots, H_n\ MHead_{\sigma_1, \ldots, \sigma_n}}$$

Table 3. Typing rules for polymorphic CHR constraints

Furthermore a type inference algorithms makes it possible to infer types for variables and for program predicates automatically. In a lattice of types with top element *term* however, the type $term \times \ldots \times term$ is always a possible type for predicates. For this reason, a heuristic type inference algorithm is used, providing a more informative type and often the expected type [9, 6]. This algorithm can also be used to infer the type of CHR constraints that are not declared by the user.

TCLP uses several solvers written in CHR. The main solver is the one for subtyping constraints. We also use a CHR solver to handle overloading of function and predicate symbols during type checking. Some other small CHR solvers are also used for handling typing environments and preliminary computations on the structure of type constructors. Hence, the possibility to type check CHR programs makes it possible that TCLP type checks its own source code.

The following example shows the typical kind of errors detected by TCLP:

Example 4. The following solver handles counters. The constraint `cpt/2` associates the name of the counter to its value, and has type $atom \times int$.[3] The constraint `val/2` also has type $atom \times int$ and constraints `incr/1` and `init/1` have type *atom*.

```
init(C) <=> cpt(C,0).
cpt(C,V) \ val(C,X) <=> X=V.
incr(C), cpt(V,C) <=> V1 is V+1, cpt(C,V1).
```

The type checker produces the following message:

```
! Error in "count.pl", line 3 :
  Incompatible types for C : atom and int
```

It is in fact an argument inversion: in the head of the last rule, the arguments of the constraint `cpt` were inverted.

The following example shows the result of type inference on a small solver:

[3] The type *atom* corresponds to Prolog atoms, that is symbols of arity 0, and not to the logical atoms.

Example 5. The following solver, taken from [17], computes the greatest common divisor of two numbers.

```
gcd(0) <=> true.
gcd(N) \ gcd(M) <=>
   N=<M | L is M mod N, gcd(L).
```

The type checker infers the following type:

```
:- typeof gcd(int) is chr_constraint.
```

that is `gcd` has type *int*.

Performance The speed of the type checker has been evaluated on ten CHR solvers taken from [17], on the solver for subtyping constraints, on the solver for overloading in TCLP, as well as on the complete TCLP source code. These tests were run on a 2 Ghz Pentium IV with 512 Mo of RAM, using the Sicstus Prolog implementation of TCLP for which the working memory space is limited to 256 Mo. The results are presented in table 4.

			Type check		Type inference	
Program	# lines	# rules	CHR	Total	CHR	Total
gcd	10	2	0.03 s	0.03 s	0.04 s	0.04 s
varleq	30	4	0.04 s	0.26 s	0.07 s	0.43 s
bool	173	78	1.32 s	2.13 s	4.63 s	5.96 s
listdom	73	13	0.78 s	1.45 s	1.77 s	2.75 s
interval	145	24	3.41 s	3.5 s	8.93 s (99.58 s)	9.03 s (99.69 s)
domain	266	84	4.30 s	6.42 s	5.35 s (183.92 s)	7.75 s (186.94 s)
fourier-gauss	328	30	1.98 s	5.88 s	6.01 s (19.04 s)	16.16 s (30.42 s)
arc	47	2	0.14 s	0.81 s	0.23 s	1.09 s
allenComp	495	490	17.48 s	17.51 s	NA	NA
subtyping	595	57	4.52 s	6.22 s	9.96 s (319.66 s)	15.28 s (322.64 s)
overloading	465	10	0.43 s	3.99 s	1.10 s	8.01 s
TCLP	4594	82	5.22 s	53.97 s	26.61 s (416.08 s)	96.09 s (518.39 s)

Table 4. Performance

The first column indicates the CLP+CHR program. The second column indicates the number of lines of codes in the program and the third one indicates the number of CHR rules in the program. Next, in column "Type checking", the type checking times are given with type inference for variables, but without type inference for predicates or CHR constraints. Finally, the column "Type inference" indicates the times for inferring types to predicates and constraints. The typing times for CHR rules are given in columns "CHR", while the typing times for the whole CLP+CHR programs are given in the column "Total". The times given between parenthesis are obtained without breaking connected components as explained in the following.

The type checking times without type inference for predicates and constraints show that the type checker is usable in practice. For example, it takes less than 18 s to check the 490 rules of the `allenComp` solver, or less than 54 s to check about 4600 lines of code constituting the source of TCLP.

In presence of subtyping, type inference needs 71 times more CPU time than type checking. In the case of `allenComp`, type inference even fails by lack of memory due to the restriction to 256 Mo. This is due to the fact that, when inferring the type of a constraint, the type checker must consider at the same time all the rules and clauses in a same connected component of the call graph, while type checking can be done rule by rule. CHR solvers often use large connected components however. One reason for this difficulty is that a few constraints used as data structures, appear in the head of numerous rules, thus creating large connected components. For example, the solver for subtyping constraints has a connected component of 54 predicates and CHR constraints. Such connected components thus require to deal with a very large number of subtyping constraints and overloaded symbols at once. Moreover, algorithms for solving subtyping constraints and overloading are potentially exponential [8, 5]. From this point of view, the performance of type inference are quite satisfactory.

It is possible to reduce type inference type by breaking such connected components. This can be done by providing the type of the constraints that are used as data structures. This technique appear to be very efficient, reducing the time for type inference in `domain` from 184 s to 5.3 s, just by giving the type of one constraint. When no time is given between parenthesis, it means that the solver was already well stratified and thus didn't need the type for some CHR constraint to be given. Moreover, type inference can be used the first time a solver is written, the inferred types being used afterwards as declarations during the rest of the development of the solver.

6 Conclusion

We have presented a type system for the *Constraint Handling Rules* CHR language [1], parameterized by the type system of the host language. In the particular case of constraint logic programming, its combination with the prescriptive type system TCLP [9] for CLP languages has been presented. Under the assumption that the well-typedness of native constraints is preserved by logical equivalence, the type system has been proved consistent w.r.t. the operational semantics of CHR and CLP+CHR respectively.

The type system for CLP+CHR is implemented as an extension of the TCLP software [10]. The reported experimental results on ten CHR solvers plus TCLP itself show that the system is already usable and useful.

As for future work, we plan to get some practical experience from the users of the system, in particular for the development of complex modular [18] and/or collaborative CHR solvers. It would also be interesting to study the instantiation of the type system with the one of Java in the framework of the JACK toolkit implementation of CHR [3] and as well as with the Haskell implementation [4].

References

1. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37** (1998) 95–138
2. Holzbaur, C., Frühwirth, T.: A Prolog Constraint Handling Rules compiler and runtime system. Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules **14** (2000)
3. Abdennadher, S., Krämer, E., Saft, M., Schmauss, M.: JACK: A Java Constraint Kit. In: Electronic Notes in Theoretical Computer Science. Volume 64. Elsevier (2000)
4. Chin, W.N., Sulzmann, M., Wang, M.: A type-safe embedding of constraint handling rules into Haskell. Technical report, National University of Singapore (2003) `http://www.comp.nus.edu.sg/~sulzmann/chr/hchr/hchr-tr.ps`.
5. Coquery, E., Fages, F.: Subtyping constraints in quasi-lattices. In Pandya, P., Radhakrishnan, J., eds.: Proceedings of the 23rd conference on foundations of software technology and theoretical computer science, FSTTCS'2003. Lecture Notes in Computer Science, Mumbai, India, Springer-Verlag (2003)
6. Coquery, E.: Typage et programmation en logique avec contraintes. PhD thesis, Université Paris 6 - Pierre et Marie Curie (2004)
7. Stuckey, P.J., Sulzmann, M.: A theory of overloading. In Peyton-Jones, S., ed.: Proceedings of the International Conference on Functional Programming, ACM Press (2002) 167–178
8. Coquery, E., Fages, F.: Tclp: overloading, subtyping and parametric polymorphism made practical for constraint logic programming. Technical Report RR-4926, INRIA Rocquencourt (2002)
9. Fages, F., Coquery, E.: Typing constraint logic programs. Journal of Theory and Practice of Logic Programming **1** (2001) 751–777
10. Coquery, E.: TCLP (2003) `http//contraintes.inria.fr/~coquery/tclp/`.
11. Abdennadher, S.: Operational semantics and confluence of constraint propagation rules. In: Proceedings of CP'1997, 3rd International Conference on Principles and Practice of Constraint Programming. Volume 1330 of Lecture Notes in Computer Science., Linz, Springer-Verlag (1997) 252–266
12. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany, ACM (1987) 111–119
13. Mycroft, A., O'Keefe, R.: A polymorphic type system for Prolog. Artificial Intelligence **23** (1984) 295–307
14. Lakshman, T., Reddy, U.: Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In Saraswat, V., Ueda, K., eds.: Proceedings of the 1991 International Symposium on Logic Programming, MIT Press (1991) 202–217
15. Smaus, J.G., Fages, F., Deransart, P.: Using modes to ensure subject reduction for typed logic programs with subtyping. In: Proceedings of FSTTCS '2000. Number 1974 in Lecture Notes in Computer Science, Springer-Verlag (2000)
16. Pottier, F.: A versatile constraint-based type inference system. Nordic Journal of Computing **7** (2000) 312–347
17. Frühwirth, T., Schrijvers, T.: (CHR web page) `http//www.cs.kuleuven.ac.be/~dtai/projects/CHR/`.
18. Haemmerlé, R., Fages, F.: Closures are needed for closed module systems. Technical Report RR-5575, INRIA (2005)

A High Performance CHR$^\vee$ Execution Engine

Luis Menezes (lcsm@upe.poli.br), Jairson Vitorino (jairson@gmail.com),
Marcos Aurelio (maurelio1234@gmail.com)

Departamento de Sistemas Computacionais**
Escola Politécnica de Pernambuco
Rua Benfica, 455 CEP: 50.720-001
Madalena, Recife- PE - Brasil

Centro de Informatica
Universidade Federal de Pernambuco
Caixa Postal 7851 CEP 50732-970
Recife - PE - Brasil

Abstract. Constraint Handling Rules with disjunction (CHR$^\vee$) is a high-level language for writing constraint solvers either from scratch or by modifying existing solvers. In this work we present a proposal for a new rule execution strategy of CHR$^\vee$ taking advantage of data structures more adequate to treat disjunctions within rules. We then compare this new approach with existing CHR$^\vee$ implementations.

1 Introduction

Over the last two to three decades, the area of Constraint Satisfaction problems and Constraint Solving has seen a great deal of developments both on the general methods for efficiently solving a broad number of problem categories and on constraint representation languages.

Constraint Logic Programming, a sub-category of constraint languages combines the advantages of logic programming and constraint solving. In logic programming, problems are stated in a declarative way using rules to define predicates. Problems are solved by an inference engine, usually using backtrack search to explore other possible solutions. In constraint solving, efficient special-purpose algorithms are employed to solve sub-problems involving distinguished relations referred to as constraints. A constraint solver can thus be seen as an inference engine as well.

Constraint Handling Rules (CHR)[8] is a declarative constraint-based programming language that has constraints as first-class concept. With CHR, any first-order constraint theory and consistency algorithm can be implemented at a high-level of abstraction.

** The Research in this paper was supported by the Brazilian Council of Research (CNPQ) under grant no. 30.0172/2004.4

CHR$^\vee$[7] is an extension of CHR that allows disjunctions making it possible to subsume the expressive power of Horn clauses program with SLD resolution [6], as is the case with Prolog.

In this work we describe a proposal for the design of a new rule execution engine for CHR$^\vee$. This engine uses a tree structure to represent the CHR$^\vee$ program execution that enables the discovery of optimizations to improve the program performance.

The rest of this paper is organized in the following sections:

- The Section 2 presents a very brief description of CHR and CHR$^\vee$
- The Section 3 motivates our approach;
- The Section 4 shows our proposal for an optimized rule execution strategy;
- The Section 5 shows a benchmarking that compares the performance of our engine with other two CHR implementations: Eclipse [1] and SICStus Prolog [3];
- Finally, the Section 6 gives some conclusions of our work.

2 Constraint Handling Rules with Disjunction

Constraint Handling Rules (CHR)[8] is a high-level language for writing constraint solvers either from scratch or by modifying existing solvers. It is a committed-choice language consisting of guarded rules with multiple heads. There are two kinds of rules: Simplification rules rewrite constraints to simpler constraints while preserving logical equivalence (e.g. A $\leq$B,B $\leq$A $\Leftrightarrow$A=B). Propagation rules add new constraints, which are logically redundant but may cause further simplification (e.g. A $\leq$B,B $\leq$C $\Rightarrow$A $\leq$C). In CHR rules, heads must be user-defined constraints, guards must be built-in constraints, i.e. constraints whose solvers are embedded in the basic CHR engine provided by the implementation, and the body may have either user-defined or built-in constraints.

Repeated application of the rules incrementally solves constraints [6]. CHR has a well-defined declarative and operational semantics and desirable properties such as soundness and completeness for its programs. Also it was shown that there is a decidable, sufficient and necessary syntactic condition for confluence of terminating CHR programs [5] and [6]. Confluence is an essential syntactical property of any constraint solver, it ensures that the solver will always compute the same result for a given set of constraints independent of the order in which rules are applied [6].

CHR$^\vee$[7] is an extension of CHR, which allows disjunctions in rule bodies. A disjunction constraint (represented by the infix operator ;) takes two constraints and specifies that at least one of them should be satisfied by a valid answer. When a CHR$^\vee$ processor finds a disjunctions in the program it tries each possible choices to find a possible answer. It is shown that CHR$^\vee$ can be used as a platform for several declarative paradigms: logic programming with integrity constraints, deduction, abduction, induction, model generation and top-down evaluation with tabulation [6].

2.1 CHR$^\vee$ by Example: the N-Queens Problem

In chess, a queen can move horizontally, vertically, or diagonally within the limits of the chessboard. The standard N-Queen's problem asks how to place N queens on a NxN chessboard so they cann't attack each other. A solution in CHR$^\vee$ is showed below (for the sake of space we model the 3-Queens problem):

```
na(X,Y,D) <=> grounded(X), grounded(Y) | P=Y+D, M=Y-D, ne(X,Y),
              ne(X,P), ne(X,M).
q(X) <=> (X=1 ; X=2 ; X=3 ).

ne(X,Y) <=> X=Y | false.

main() <=> q(X1), na(X1,X2,1), q(X2), q(X3),
           na(X1,X3,2), na(X2,X3,1).
```

The first two lines of the solution are the actual CHR$^\vee$ program. It consists of two rules: the first rule defines de constraint `na` (short for non-attack), the guard expression of the rule states that `X` and `Y` must be grounded. The rule body specifies that two queens located in different lines cannot be in the same column neither in the same diagonal.

The second rule states that a queen can take any of 3 values, the semicolon meaning disjunction. Note that the guard for this rule is simply true because there is no pre-condition needed to fire the rule.

The third rule defines the constraint `ne` behavior, this constraint sets that two arguments should be different and its definition just aborts the current solution set if it can prove that the arguments has the same value.

The `main()` constraint is a reserved constraint that indicates a query on the given CHR$^\vee$ program. It states that `X1`, `X2` and `X3` are positions of the queens placed in line 1, 2 and 3 and it describes the non-attack condition in terms of the vertical distance between the combinations of any pair of queens. For example na(X1,X2,1) means that the queen in line 1 and the queen in line 2 have a vertical distance of 1 and the non-attack constraint will enforce that they will not be neither in the same column nor in the same diagonal. The other constraints work the same way for the other pairs.

The CHR engine starts by adding all the query constraints in the Constraint Store, here we assume that the engine consumes the constraints from left to right, but this may vary accordingly to the implementation. As soon as all the constraints are stored the program picks the first one and begins to scan the CHR program. In our case the first constraint would be q(X1) which fires rule 2, adding the built-in constraint X1=1 to the constraint store. The second constraint tried is na(X1,X2,1), which cannot fire at first because X2 is not grounded yet, hence this constraint is suspended for now and the next constraint is tried. Whenever a built-in is added to the store the CHR$^\vee$ engine searches for suspended constraints, which might fire thanks to the new built-in constraint added. When the next constraint q(X2) fires it adds the built-in X2=1 in the store. This time na(X1,X2,1) is fired and it adds all the constraints on the right-side of

rule 1. Namely P=Y+D, M=Y-D, ne(X,Y), ne(X,P), ne(X,M). The CHR engine finds then that ne(X,Y) cannot be satisfied because X1 and X2 both have the value 1, hence it backtracks to the last added constraint, which was X2, and tries another value for X2, namely 2. This process will go on until the constraint store is empty or there are no more rules to apply to the suspended constraints.

If we look closely to this execution strategy we shall notice that for constraints in the form: `(a ; b) , c`, this is executed similarly as the sentence: `(a , c) ; (b , c)`, whenever the engine backtracks it typically has to process the constraint c again, which can be rather expensive if there are many rules with disjunctions. In the example above in particular, `na(X1,X2,1)` is processed every time a new value for `X2` is tried. To better illustrate our point consider another simple CHR program:

```
dummy(X,Y) <=> grounded(X), grounded(Y) | gt(X,Y+2).
q(X) <=> true | (X=1 ; X=2 ; X=3; X=4 ).
```

And consider the query:

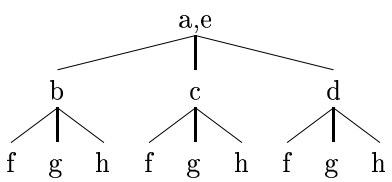
```
main() <=> true | q(X1),dummy(X2,X1),q(X2).
```

The traditional CHR$^\vee$ engine execution with backtracking will fire the constraint dummy for all the choices of: `X2=2, X2=3`, and: `X2=4` when it finally manages to reach a solution. In general current CHR$^\vee$ implementations are unable to handle efficiently states like: `a,(b;c),f` , where the constraint `a` and `f` can be processed by some CHR-rule (R1) independently of constraint introduced in the disjunctions. When this happens, the effects of R1 execution are reversed and redone for all disjunctions combinations. In the following sections we present a strategy for solving this problem.

3 A New Execution Environment for CHR$^\vee$

In this paper we propose a new execution engine for CHR$^\vee$ programs that tries to identify when the rule application is independent of the ordering of disjunctions and can avoid some redundant rule execution by preserving these rules effects when a backtrack is performed.

In our proposal, the constraint store of the CHR$^\vee$ program is represented as a tree, whose branches indicate constraints produced by the program's disjunctions. For example, the formula a,(b;c;d),e,(f;g;h) can be represented by the following tree structure:



The tree nodes are made up a set of marked constraints using the following auxiliary notations:

- c* (unprocessed constraint). This means that no search was performed to find some CHR$^\vee$ rule that handle the constraint c;
- $\underline{\text{c}}$ (executing constraint). Indicates that this constraint is about to be processed by the engine;
- c_n^b (backtracked constraint). This means that the constraint "c" is not part of the constraint store but was removed from a parent node (located on depth n) and should be reinserted when the node is removed from the tree;
- c (unmarked item): Indicates a suspended constraint meaning that a rule match was already tried with no success. When this happens the constraint will be stored in the tree node until that new constraints added to the Constraint Store allow a match;

3.1 Executing a CHR$^\vee$ Program

Let us now describe the proposed rule execution strategy by looking only at a query statement without bothering with a real CHR$^\vee$ program. Consider the query: A,(B;C),D, (E;F). Initially the engine defines a tree formed by a single node without children:

A*,(B;C)*,D*,(E;F)*

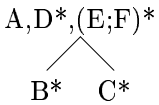
At first all constraints are unprocessed and the engine chooses the first one (from left to right) to be executed:

$\underline{\text{A*}}$,(B;C)*,D*,(E;F)*

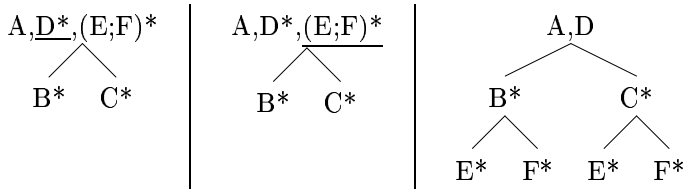
The constraint A is then unmarked and the next one is picked:

A,$\underline{\text{(B;C)*}}$,D*, (E;F)*

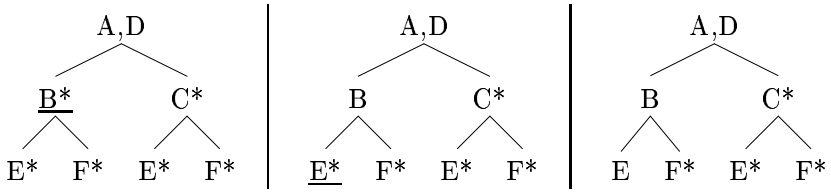
When the disjunction is found, it is removed from the node and two children nodes representing the disjunction will be inserted as its leafs:



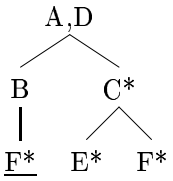
Now, the CHR$^\vee$ processor will continue to process the constraints located in the root node:



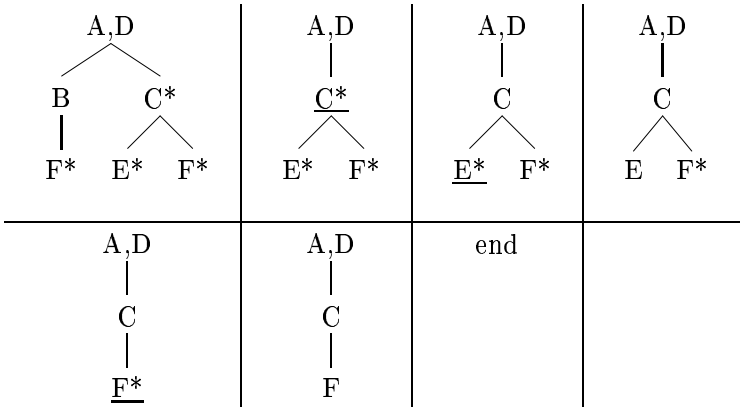
When all constraints in root node are processed, the engine will traverse the tree using a depth-first search strategy taking always the left most node:



A solution is found when a leaf is reached and processed. It then is presented to the user. If the user requires another solution, the engine will move up through the tree (removing the visited node) until it finds a node with a non-visited child:



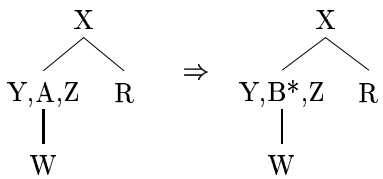
This process continues until all tree nodes are removed:



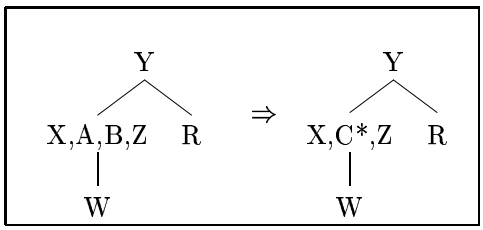
3.2 Executing Unconditional Rules

When a constraint is processed it can fire a CHR-rule, when this happens constraints can be removed from the constraint store and new ones will be inserted. Now we will describe how these effects are represented in the tree notation.

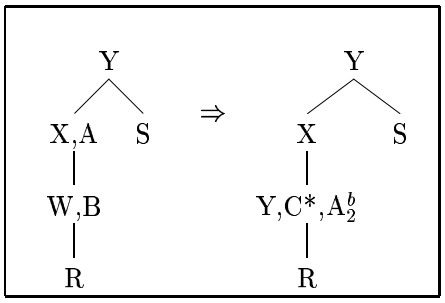
The simplest kind of simplification rule: A $\Leftrightarrow$ B, just replaces the constraint "A" for the constraint "B". When this rule is processed, the replacement is performed in the tree node:



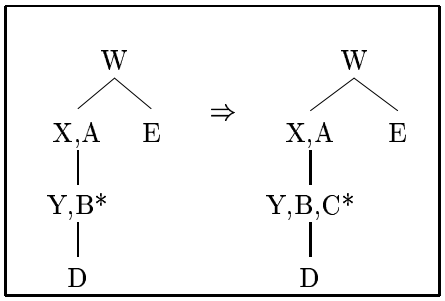
If the rule head contains more than one constraint: A,B ⇔ C, there are two different scenarios: "A" and "B" can be in the same tree node or in different tree nodes. If they are in the same node they are simply replaced by the constraint C:



However, if they are located in different tree nodes, the constraint located in the upper node are removed from the tree and the constraint located in the lower node is replaced by "C", additionally, a backtrack constraint is inserted in this node to guarantee that the constraint "A" will be reinserted in the tree if this branch is removed:



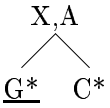
Propagation rules behaves like simplification rules: A,B ⇒C, inserts a new constraint C in the current visited tree node where the constraints A and B are. As showed bellow, in our approach the new constraint is placed only in the deepest node containing the constraints A and B:



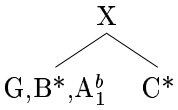
3.3 Executing Conditional Rules

When a rule contains a guard (G) that depends on built-in constraints: A,B ⇔ G | C; this rule can be fired in two different moments: when a constraint A or B is placed in the constraint store or when the guard becomes true. If the first case

happens, the rule execution is performed as an unconditional rule. Otherwise, if the user defines a new constraint that activates the guard, the rule is executed considering this constraint as part of the head rule. The next example illustrates this occurrence:

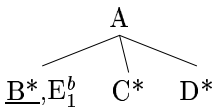


if the rule A $\Leftrightarrow$G | B is part of the program. The execution of "G" will enable this rule and when it is executed, the constraint "A" will be replaced by "B". However in this case, the guard "G" is the responsible for the rule execution and it is handled like any constraint belonging to the rule Head and the resulted program is:

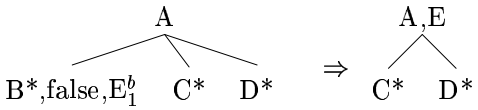


3.4 The Predicate Fail

The predicate "fail" indicates the impossibility to find a solution in the current branch. When a rule inserts this constraint in some branch, it will be removed from the tree (all registered backtracks are restored) and another branch is processed. For example:

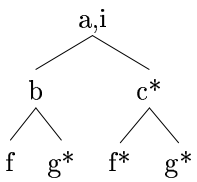


If the rule A,B $\Rightarrow$fail is executed, the constraint false is inserted in the branch, causing its removal:

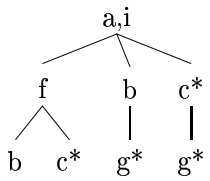


3.5 Optimizing the Execution

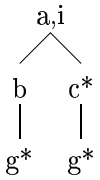
Using this structure to model the execution of CHR$^\vee$ programs it is easy to find and to take advantage of some constraints dependencies to improve the program execution performance. For example the execution of: `a,(b;c),(f;g),i`, will produce the following tree:



A possible optimization would be: If the constraints "a" and "i" can be replaced by the "x" independently of another tree nodes, this action is executed once for all six leafs. The hierarchical tree structure allows the identification of new optimization techniques for example: If the processing of "f" occurs independently of "b", this results would be the same produced by the second evaluation of "f" in the other tree branch. To avoid the redundant rule execution, the execution tree can be rewritten to a more efficient equivalent structure:



Specially, if "f" is reduced to "false", the execution tree could be further reduced to:



4 Performance Evaluation

To evaluate the results of the proposed execution model, we developed a partial implementation in the Java programming language and measured the execution time to solve the following problems:

1. The N-Queens problem: that finds how to place N chess queens on a NxN board in a configuration where no queen can attack another. We tested this example using the following values for N: 8,10,12;
2. SAT problems: that tries to find all possible variable assignment that satisfies a propositional logic predicate. We used some standard test cases available in [9]. We show the results obtained with the cases coded: uuf-100-0101 and aim50-20-no1.

Table 1 shows the results obtained by using two versions of the engine: with the tree rewriting optimization proposed in the Section 3.5 turned off and on (the optimized implementation was not applied to the SAT problem). The results are compared with other CHR$^{\vee}$ engines embedded in Eclipse Prolog [1], SICStus Prolog [3] and XSBProlog [4]. The tests were performed on a Pentium III machine with 750MHz running the Linux Debian operating system.

The results shows a significant performance gain obtained with our implementation (about 5-15 times faster) and the use of an optimized engine reduces the execution time about 40

	Java	Opt Java	Eclipse	XSB Prolog	SICStus
queens8	.38s	.30s	0.52s	2.24s	3.33s
queens10	.93s	.648s	14.31s	68.41s	79.95s
queens12	28.9s	17.3s	510s	+1000	+1000
uuf100-0101	19.05s	X	653.32s	ERROR	+1000s
aim50-20-no1	12.92s	X	61.61s	ERROR	649s

Table 1. Performance Tests

5 Related Work

The work of Wolf[10], has a motivation similar to our work. He proposes an adaptive CHR execution engine that allows the arbitrary sequences of constraint additions and deletions, which are necessary for intelligent search strategies like conflict-directed back jumping or dynamic backtracking. In this context, "adaptive" means that constraint additions and deletions in arbitrary order are supported, i.e. after each change of the considered constraints, the derivations based on CHR are adapted accordingly. For example, if there is a rule `leq(X,Y), leq(Y,Z)` $\Rightarrow$ `leq(X,Z)`, and a query `leq(A,B), leq(B,C)`, this will propagate `leq(A,C)`. However the deletion of the constraint `leq(A,B);leq(B,C)` causes the derived constraint `leq(A,C)` to be deleted, too (leq here stands for lesser than or equal). This approach have a clear advantage which is to allow in deletions and additions of any constraints and not only in the disjunctions occurrences as is in our case. However the proposal presented here uses far less memory than Wolf[9] as we do not use justfications for each rule (a special data structure that tracks which constraints were generated by previous ones).

6 Conclusions and Future Work

This works defines a new CHR$^\vee$ rule execution strategy that takes advantage of a more elaborated search process to improve the execution performance. These initial test results obtained are very promising to use the proposed engine as the core of a high perform CHR$^\vee$ engine.

This study is part of project ORCAS[2] that aims to improve the usability of constraint programming by defining new applications and constraint based tools.

The next part of this work is to define tree transformations to further improve performance and to prove their soundness.

References

1. Eclipse constraint logic programming system. Website http://www.icparc.ic.ac.uk/eclipse/.
2. ORCAS Website http://www.cin.ufpe.br/ orcas/.

3. SICStus Prolog. Website http://www.sics.se/sicstus/products/.
4. XSB Prolog. Website http://xsb.sf.net.
5. Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. *Third International Conference on Principles and Practice of Constraint Programming*, 1997.
6. Slim Abdennadher. Rule-based constraint programming: Theory and practice. 2001. Habilitationsschrift.
7. Slim Abdennadher and Heribert Schütz. Chrv: 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.
8. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 1998.
9. Holger H. Hoos and Thomas Stützle. Satlib: An online resource for research on SAT. *SAT 2000*, pages 283–292, 2000. SATLIBis available online at http://www.satlib.org.
10. Armin Wolf. Adaptive solving of equations over rational trees. *Lecture Notes in Computer Science*, 1520, 1998.

K.U.Leuven JCHR: a User-Friendly, Flexible and Efficient CHR System for Java

Peter Van Weert, Tom Schrijvers*, Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium

Abstract. A new integration of Constraint Handling Rules (CHR) and Java is presented: the K.U.Leuven JCHR system. It was designed with three aims in mind: user-friendliness, flexibility and efficiency.
User-friendliness is achieved by providing a high-level syntax that feels familiar to both Java programmers and users of other CHR embeddings, and by full compliance to the refined operational semantics. Flexibility is the result of a well thought-out design, allowing e.g. an easy integration of built-in constraint solvers and variable types. Efficiency is achieved through an optimized compilation to Java and the use of a very efficient constraint store. The design of the K.U.Leuven JCHR system is presented, its implementation discussed and its performance is compared with other CHR systems embedded in Java and Prolog.

1 Introduction

Constraint Handling Rules (CHR) is a high-level, declarative programming language, originally designed for the introduction of user-defined, application-tailored constraint solvers in a given host language. Nowadays CHR is increasingly being used as a general programming language in a wide range of applications – multi-agent systems, type systems and natural language processing, to name some – and for the efficient implementation of classical algorithms.

Since its conception in 1991, CHR systems have been developed for several host languages, with the main emphasis being placed on Prolog implementations. The first full CHR system [16] was developed by Christian Holzbaur in co-operation with Thom Frühwirth. This system is included in SICStus Prolog and Yap and is considered the reference implementation. In the past five years CHR systems have been written for HAL and Haskell. The most recent and advanced CHR system for Prolog is the K.U.Leuven CHR system [21,20], originally hosted by hProlog, but meanwhile ported to two major open-source Prolog systems: SWI-Prolog and XSB.

Another programming language that saw the light in 1991 is Java. This platform-independent, object-oriented language has become very popular over the years. There is a growing interest to extend Java with constraint handling

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

support for implementing applications like planning and scheduling. For this purpose several specific constraint solvers have been implemented in Java [8,25,24]. These so-called black-box solvers require substantial effort in program development and the resulting programs are hard to maintain, modify and extend. A user-friendly and efficient CHR system for Java would considerably simplify the design of new application-tailored constraint solvers for Java, and it would also enable the introduction of the numerous solvers already written in CHR into Java.

Two (prototype) CHR systems already existed in Java at the moment our research started. The first, JCHR,[4] is reasonably user-friendly, but lacks efficiency as we will demonstrate in Section 4. The second, DJCHR [27], is more efficient, but fails to satisfy the user-friendliness requirement. Neither system truly integrates the CHR language into Java, but rather introduces a form of logic programming (LP) on top of Java. Interaction with constraints other than equality is not possible, nor does the design allow for an easy incorporation of new optimizations. The goal we set ourselves in designing and implementing a new system is therefore threefold:

1. **User-friendliness** The high-level syntax used should feel familiar to both CHR and Java adepts. The host language has to be Java (and not "LP in Java"), so one should be able to use regular Java-variables and arbitrary Java-expressions. Also, we must not lose track of the declarative nature of CHR and Constraint Programming (CP): programming in terms of CHR rules and constraints (both built-in and user-defined) should be fully supported. Moreover, to allow an easy porting of existing solvers, support for the same familiar semantics is at least as important as a similar syntax.
2. **Flexibility** Adding of and experimenting with new built-in solvers, built-in constraints, static program analyses and optimizations should be possible. Therefore a well thought-out design is indispensable.
3. **Efficiency** The CHR constraint handlers should be efficient, preferably competitive with their Prolog counterparts. For this goal one can rely on the wide experience acquired with the existing Prolog and HAL systems: several techniques and optimizations can be tailored to the new host language.

Overview We assume the reader to be acquainted with Java, Constraint (Logic) Programming [7,12] and the syntax and semantics of Constraint Handling Rules [11]. Some familiarity with the implementation of CHR in Prolog systems [16] is also helpful. In Section 2 we start with a brief review of some general principles, after which we provide an overview of the two pre-existing Java implementations and their respective shortcomings. Next, Section 3 discusses the K.U.Leuven JCHR System in detail. The performance of the K.U.Leuven JCHR System is compared with that of several other systems in Section 4. Finally, Section 5 concludes.

This paper is based on work done in the context of the master thesis of the first author [26] (in Dutch), in which a more comprehensive exposition can be found.

2 Implementation of CHR: Related Work

2.1 General Principles

Following the well-known reference implementation by Christian Holzbaur [16], a typical CHR system consists of two parts:

- A *compiler* that compiles CHR rules into code of the host language. For a survey of the compilation scheme used in the reference implementation we refer the reader to [20].
- A *runtime (environment)* containing generic code that is reused by all CHR programs. The most important component of the runtime is the constraint store, providing efficient lookup of partner constraints and reactivation of user-defined constraints.
  Because the genericity of the runtime code has its price, namely runtime overhead, its role is more and more taken over by specialized code generated by optimizing compilers.

A CHR implementation has to follow the semantics of the language. Besides the declarative semantics [11] (which lets a CHR program easily be understood without knowing how it is executed), several versions of operational semantics can be found throughout the literature. The theoretical operational semantics [2] is highly non-deterministic and therefore still too high-level to fully describe the behavior of most current CHR implementations. The refined operational semantics, formally defined in [10] and implemented in all major CHR systems, remedies this shortcoming by establishing a partial order of execution and treating CHR-constraints as procedure calls.

2.2 JCHR (JaCK)

The Java Constraint Kit (JaCK) [4] is a library for constraint programming (CP) in Java. It is composed of:

JCHR: Java Constraint Handling Rules (JCHR) [19] is the CHR dialect with matching compiler and runtime we will focus on shortly.

VisualCHR: An interactive tool visualizing the execution of JCHR [5].

JASE: CHR is often not capable of solving non-trivial CP problems unless combined with search. Java Abstract Search Engine (JASE) [17] is a framework designed to work in tandem with JCHR. As soon as JCHR is no longer able to do any work, JASE kicks in and makes a choice about which direction a solution is to be searched in. When JCHR detects an inconsistency JASE backtracks to the most recent choice-point. JASE is considered abstract because it allows for the specification of the search-strategy.

The JCHR syntax, illustrated in Fig. 1, is a combination of CHR and Java [19]

> [...] intended to be very similar to Java, in order to provide an intuitive programming.

```
handler leq {
    class java.lang.Integer;

    constraint leq(java.lang.Integer, java.lang.Integer);

    rules {
        variable java.lang.Integer X, Y, Z;

        if (X = Y)   { leq(X, Y) }  <=> { true }      reflexivity;
        { leq(X, Y) && leq(Y, X) }  <=> { X = Y }     antisymmetry;
        { leq(X, Y) &\& leq(X, Y) } <=> { true }      idempotence;
        { leq(X, Y) && leq(Y, Z) }  ==> { leq(X, Z) } transitivity;
    }
}
```

Fig. 1. A leq-handler in JaCK's JCHR

In pursuit of this goal the authors have made some changes to the original CHR syntax that, in our opinion, are unnecessary or even counterintuitive. We will elaborate further on these and other syntactic elements when we contrast them with our approach in Section 3. The variables used by JaCK are exclusively logical variables. The most important conceptual difference with a Prolog system is that they are typed (meaning they can only be bound to variables or values of the same[1] type), necessitating variable declarations. Likewise arguments of user-defined constraints are also typed, leading to extended constraint-declarations (cf. Fig. 1 for an example). The only built-in constraint known (and possible!) is equality, which appears to be implemented with an unoptimized version of the union-find algorithm [22,13]. Besides user-defined constraints, equality and logical variables, a very limited amount of Java-like expressions can be used. Some observations:

- In JCHR one cannot use a variable as implicit argument (as in X.someMethod()), since [19]

  > [...] the programmer is forced to use methods, which [take the variables as parameters and] return the result of their computation. [...] It is a question of design, whether to provide this functionality or not. [JCHR] does not implement this property of Java variables in JCHR variables, in order to stay closer to the properties of logical variables.

- Some simple syntactic sugar is used for the following literals: integer-, float- and string-objects. For example a "1" always gets translated into "new Integer(1)". The undesirable result is that primitive values can *never* be used.

The so-called JCHR-Compiler is actually merely a tool that translates a JCHR program to high-level, unoptimized Java-code. As [3] states:

[1] Unfortunately in the current implementation: *exactly* the same type...

> [...] it is possible to create rules directly in Java. But the compiler is a useful tool, if CHR is already known from Prolog.

Note that this last statement is questionable, given the numerous syntactical changes that were introduced. Later on this intermediate, "compiled" form is interpreted by the runtime (called the JCHRCore-Evaluator). The fact that virtually all code is generic results in high runtime penalties, which can be avoided when compiling to specialized, lower-level code. The only real effort in making the execution more efficient seems to be the use of path indexing [23] for partner constraint lookup. Furthermore, the interpreter does not follow the refined operational semantics. Even worse maybe is that the implemented semantics is not even completely in compliance to the theoretical operational semantics. The former shortcoming leads to the observation that programming can become rather impractical or counterintuitive, whilst the latter actually means that strictly speeking JCHR isn't even a form of CHR.

2.3 DJCHR

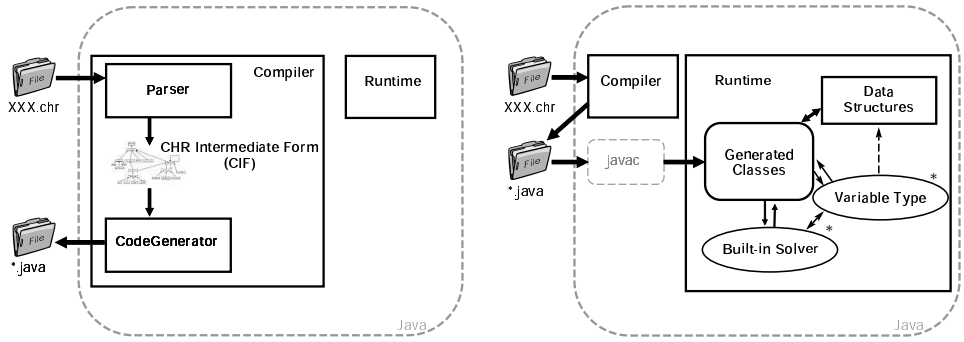
DJCHR aims at introducing a *dynamic* variant of CHR in Java. In practice, many constraint problems change continuously: arbitrary constraints can be added and removed at any time. After each change the solution has to be adapted, rather than solved from scratch. The fact that such applications are typically used online, makes requirements as efficiency and stability all the more important.

A dynamic version of CHR lets all existing CHR handlers be used in dynamically changing environments. Whilst incremental addition of constraints is already characteristic of CHR, to support arbitrary deletions a new incremental operational semantics had to be devised [28].

Syntactically, DJCHR [27] completely follows logic programming: logical variables and compound terms are supported, as are some numerical constants. The only implemented built-in constraint is term equality, using an adaptive unification algorithm. Unfortunately, DJCHR handlers can only be coded directly in Java-code, using expressions similar to the ones JaCK translates its high-level, declarative JCHR programs to. On the other hand, DJCHR *does* compile these Java-coded rules to more efficient, low-level Java-code. The compilation scheme and runtime system are considerably different from other CHR systems because of the dynamic requirements. Based on the accompanying documentation, we presume several optimizations are yet to be made. But given the compiler-design – a single 5000-line class – it looks like this might not always be equally straightforward.

3 The K.U.Leuven JCHR System

As most CHR systems the K.U.Leuven JCHR system consists of two main components: a compiler and a runtime. An overview of the architecture can be found in Figure 2.



(a) The Compiler Subsystem (b) The Runtime Subsystem

Fig. 2. High-level architectural overview of the K.U.Leuven JCHR system.

The compiler subsystem (Fig. 2(a)) takes a K.U.Leuven JCHR source file as input and outputs Java source files. Its kernel consists of a parser generated from an ANTLR-grammar and code generation driven by FreeMarker templates. In between these stages a flexible and complete model of the JCHR program is constructed, called *CHR Intermediate Form* (CIF).

The runtime subsystem (Fig. 2(b)) consists of several generic data-structures (mainly lists and iterators), built-in solvers and variable types. Together with those generated by the compiler subsystem, these generic classes make up an efficient handler capable of solving the user-defined constraints.

So, conceptually, we take the first stage of JaCK – a high-level, declarative syntax with corresponding ANTLR-generated parser – followed by the final stage of DJCHR – full compilation to low-level Java-code. But this is mostly where the analogy ends: we will show that there exists significant differences in both design-goals and their implementation. The concrete aim can succinctly be summarized as follows:

$$\text{K.U.Leuven JCHR} = \text{Java} + \text{CP} + \text{CHR} \tag{1}$$

Hereby it is fundamental that the three terms in right hand side of the equation are equally important. As seen in the previous section, the sum for both other Java-implementations leans more towards "Java + LP + CHR", with a rather inferior role assigned to the integration with Java or even to some important aspects of CHR.

3.1 User-friendliness

The high-level syntax we propose is aimed at an intuitive and familiar look and feel for users of both Java and existing (Prolog-)CHR-systems, allowing a declarative implementation of efficient constraint solvers in Java. Although we

have this goal in common with JaCK[2], there are several important distinctions. Figure 3 shows the classical leq-handler in K.U.Leuven JCHR syntax, illustrating some but not all aspects mentioned throughout this subsection. More example programs can be found on `http://www.cs.kuleuven.be/~toms/CHR/JCHR/`.

In the remainder of this subsection we will zoom in on several syntactical aspects, grouped under the three terms of Equation 1.

```
import runtime.LogicalVariable;

handler leq<T> {
    solver runtime.EqualitySolver<T>;

    constraint leq(LogicalVariable<T> X,
                   LogicalVariable<T> Y) infix =<;

    rules {
        variable LogicalVariable<T> X, Y, Z;

        reflexivity  @ X =< Y <=> X = Y | true;
        antisymmetry @ X =< Y , Y =< X <=> X = Y;
        idempotence  @ X =< Y \ X =< Y <=> true;
        transitivity @ X =< Y , Y =< Z ==> X =< Z;
    }
}
```

Fig. 3. A leq-handler in K.U.Leuven JCHR

CHR The core of a CHR handler, the declaration of the rules itself, is superficially analogous to the original CHR syntax. Contrary to what is the case in JaCK, this means that the (optional) name precedes a rule and the guard follows the rule-operator. Also, the brackets and the "if (...)"-construction have been removed. The reason is that these changes (which were introduced by JaCK to make the syntax more Java-like), in our opinion, only hinder the rapid porting of existing handlers. Sometimes they even feel counterintuitive, e.g. when placing constraints, i.e. the guard, on variables only introduced afterwards. Only the possibility to separate conjunctions with double ampersand ('&&') and to end a rule with a semicolon (';') is kept, as an alternative to the common Prolog-like notation (',' and '.' respectively). The ugly '&\&' in simpagation rules though has been discarded, in favor of '\'.

Java When looking at the conjuncts, arguments and variables we see that the K.U.Leuven JCHR language leans more towards Java than JaCK does.

[2] Henceforth we will simply use JaCK to refer to the JCHR-portion of this library, to avoid confusion with K.U.Leuven JCHR.

Java variables The introductory text [12] reminds us of the following:

> The key aspect in the CLP(X) scheme is to provide the user with more expressiveness and flexibility concerning the primitive objects the language can manipulate. Clearly the user wants to design his application using concepts that are as close as possible to his domain of discourse, e.g. he wants to use sets, boolean expressions, integers, rationals or reals, instead of coding everything as uninterpreted structures, i.e. finite trees [Herbrand terms], as is advocated in logic programming.

Since using real-world concepts is one of the cornerstones of object-oriented programming, an artificial limitation to logical variables (as is the case in DJCHR) would be a step back. The typed logical variables used by JaCK are a step in the right direction, but are still too restrictive and counterintuitive for a Java-programmer. The K.U.Leuven JCHR system therefore basically uses fully-fledged Java-variables, which in principle can have any valid Java-type, including primitive types and generic types (new in Java 1.5). Thanks to the fully supported generic types [6] we are e.g. able to use `LogicalVariable<T>` as a variable-type, substituting `T` for the type we want our logical variable to have. These variables are completely equivalent with the typed logical variables of JaCK. Important to note however is that conceptually this type in itself is no different from any other Java-type. Note that this is just one of the many possibilities: other interesting options include finite-domain variables or even complete Herbrand terms (like the ones used in DJCHR or Prolog), although some syntactic sugar might be in order here.

Java expressions These Java variables can be used in a wide range of Java-expressions, as a conjunct in guard or body or as an argument. A key difference with JaCK is that Java variables *can* be used as implicit argument.

Generic handlers To ease the transition from untyped Prolog to strongly typed Java we have introduced generic handlers. Without this feature we would e.g. have to implement a different leq-handler (cf. Fig. 3) for each variable-type required. To the best of our knowledge the K.U.Leuven JCHR system is the first typed CHR-system that adequately deals with polymorphic handlers this way. Furthermore this feature makes a handler-declaration completely analogous to a Java-class, allowing an arbitrary number of formal type parameters, each having any number of upper-bounds. For more information on these concepts and some constraints we refer the reader to e.g. [6].

Other Java-related syntactical aspects

- Since our variables are typed, they also have to be declared. We've used the same keyword as JaCK, i.e. `variable`, for this purpose. Note nevertheless that we are not declaring logical variables, as is the case in JaCK, but Java-variables!

– Whereas Prolog systems declare constraints using the keyword `constraints` followed by a comma-separated list, JaCK uses the `constraint` keyword for this, declaring each constraint separately. We support both options. As is the case of JaCK, the arguments of user-defined constraints are also typed, naturally leading to a declaration analogous to a method-signature in Java. To strengthen this analogy, but also to improve readability of debugging information and the generated code, one can also give these arguments a name in K.U.Leuven JCHR.
– An infix-notation can be used for constraints, as illustrated in Fig. 3.
– JaCK uses obligatory `class`-declarations, while still requiring to use the fully qualified name everywhere. K.U.Leuven JCHR on the other hand uses `import`s completely analogous to Java imports[3], i.e. the use is only an optional tool, but once a type is imported it is no longer necessary to use the fully qualified name. To strengthen the analogy with a Java-class even further, but also because one has to be able to use imported classes in a generic handler-declaration, these imports have been moved in front of the handler-block.

CP Besides the use of Java variables and expressions, programming in terms of constraints is fully supported. Inspired by the HAL CHR-system [9] arbitrary built-in constraints and solvers can be used: in no way do we restrict the system to equality. For this the K.U.Leuven JCHR system considers equality over logical variables (cf. JaCK) or Herbrand terms (cf. DJCHR and Prolog) "just another constraint".

A final, declarative feature offered by the K.U.Leuven JCHR system is *coercion*. For Java-programmers: this can be seen as an extension of the new auto-boxing/unboxing feature in Java 1.5. Thanks to automatic coercion one can for example simply type "`3`" instead of the cumbersome expression "`new LogicalVariable<Integer>(Integer.valueOf(3))`", or vice versa use an expression whose type is `LogicalVariable<Float>` in places where a `Float` or `float` is expected. In other words, coercion vastly improves over the simple syntactic sugar used by JaCK. Thanks to this feature the programmer seldom has to be concerned with typing (e.g. when porting from untyped Prolog) and can thus concentrate on what really matters.

3.2 Flexibility

The next important criterion is the flexibility of a system, encompassing its (re)usability, adaptability and extensibility. In general this implies adopting a well thought-out design, using among other things suitable design-patterns [14] (cf. also Fig. 2), but several other aspects are equally important. We review some in this section.

[3] Package imports and static imports are not yet implemented at the time of writing.

CP As seen in the previous subsection one can in principle use arbitrary high-level built-in constraints. This necessitates a means to inform the compiler about the Java-expressions they can be translated to. The K.U.Leuven JCHR system implicitly knows about the the special class `Comparator`-solvers and most frequently used constraints in Java, including (dis)equality-constraints over primitive variables and `Comparable` objects. For other constraints their respective solvers have to be disclosed to the system via a `solver`-declaration (as illustrated in Fig. 3). Furthermore we have chosen to use annotations, the new meta-data facility of Java 1.5, to declare which method(s) have to be used for which constraint. An example of an annotated solver-interface can be seen in Figure 4. Using extensive reflection and some heuristics the K.U.Leuven JCHR compiler is herewith capable of transforming constraint-statements into their corresponding Java-expressions (possibly coercing the arguments as well).

```
import annotations.*;

@CHRconstraints({
    @CHRconstraint(identifier = "eq", arity = 2, infix = "=")
})
public interface EqualitySolver<T> {
    @CHRtells("eq")
    public void tellEqual(LogicalVariable<T> X, T val);
    @CHRtells("eq")
    public void tellEqual(T val, LogicalVariable<T> X);
    @CHRtells("eq")
    public void tellEqual(LogicalVariable<T> X, LogicalVariable<T> Y);

    @CHRasks("eq")
    public boolean askEqual(LogicalVariable<T> X, T val);
    @CHRasks("eq")
    public boolean askEqual(T val, LogicalVariable<T> X);
    @CHRasks("eq")
    public boolean askEqual(LogicalVariable<T> X, LogicalVariable<T> Y);
}
```

Fig. 4. An annotated solver-interface.

Java Another feature reviewed in the previous subsection is the use of arbitrary variable types. Again the system is implicitly fully informed about the most frequent types, including all eight primitive types and their wrappers and `String`s, while for other types the necessary information (needed e.g. for coercion and initialization) can easily be explicitly declared using annotations.

CHR The K.U.Leuven JCHR system uses ANTLR [18] (parser-generator) and FreeMarker [1] (template-engine used for code-generation), which enjoy advan-

tages very similar to declarative programming, like readability and adaptability. The FreeMarker-templates especially have already proven their flexibility when introducing several optimizations to the compilation scheme.

The flexible intermediate form, named CIF (cf. Fig. 2(a)), introduced between parsing and code-generation is well-suited for static program analysis. Furthermore a framework has been designed to allow for an easy introduction of indices for the constraint store.

With this last subsection we are clearly already on the verge of the final requirement we will consider, namely efficiency.

3.3 Efficiency

In this subsection we will review some implemented efficiency-improvements in the successive stages of the system, ranging from the analysis of a CHR-program, over the corresponding generated code, to its actual execution. From the previous subsection we already know that it should be easy to add several other improvements. In the next section then we will put the current implementation to the test as we compare its performance with that of several other systems.

Program analysis One type of static program analysis has been implemented, which we have called scheduling. Implemented schedulers use techniques like early guard scheduling and the introduction of lookups through the appropriate indices to speed up the searching of partner constraints. This approach is designed to be easily extended towards e.g. join ordering [15].

Compiled code The compilation scheme used, though originally based on a typical Prolog-schemes [20,16], has been fully tailored to the new host language. Equivalents of all simple optimizations listed in [20] are implemented, many for the first time in a Java system. Because Java is an imperative language, the continuation-based control flow has been made significantly more sequential. Exactly this is also what we exploited by introducing a new optimization called explicit backjumping (in contrast to the implicit backtracking used in Prolog systems), possibly capable of avoiding trashing, one of the well-know drawbacks of backtracking.

As with other more mature systems, the K.U.Leuven JCHR system has seen a gradual movement from generic runtime-code towards more efficient code generated by the compiler. The constraint store for instance consists in the current version for the most part of specialized generated code.

Indices After the K.U.Leuven CHR system, the K.U.Leuven JCHR system is to the best of our knowledge the second CHR implementation allowing hash-table indices for the constraint store, speeding up lookups based on equality constraints in the guard (or implicit constraints originating from using the same variable multiple times in a head). In addition, for the first time hash-indices can be used for variables whose hash-value might change over time (in Prolog this corresponds to non-ground terms).

Built-in solvers The only non-trivial built-in constraint solver already implemented for the system handles equality constraints over logical variables (cf. Fig. 4). Its implementation is based on the union-find algorithm, optimized with path compression and union-by-rank [22,13].

4 Experimental Evaluation

In this section we compare the performance of five different CHR-systems: two Prolog-systems, the SICStus reference implementation (for short: **SICStus**) and the K.U.Leuven CHR system as ported to SWI-Prolog (**SWI**) and all three existing Java systems (**DJCHR**, **JaCK** and of course the **K.U.Leuven JCHR** system). The benchmarks used in this section can be found on `http://www.cs.kuleuven.be/~toms/CHR/JCHR/`. We were unable to port some benchmarks to JaCK and/or DJCHR, partially due to the differences in semantics.

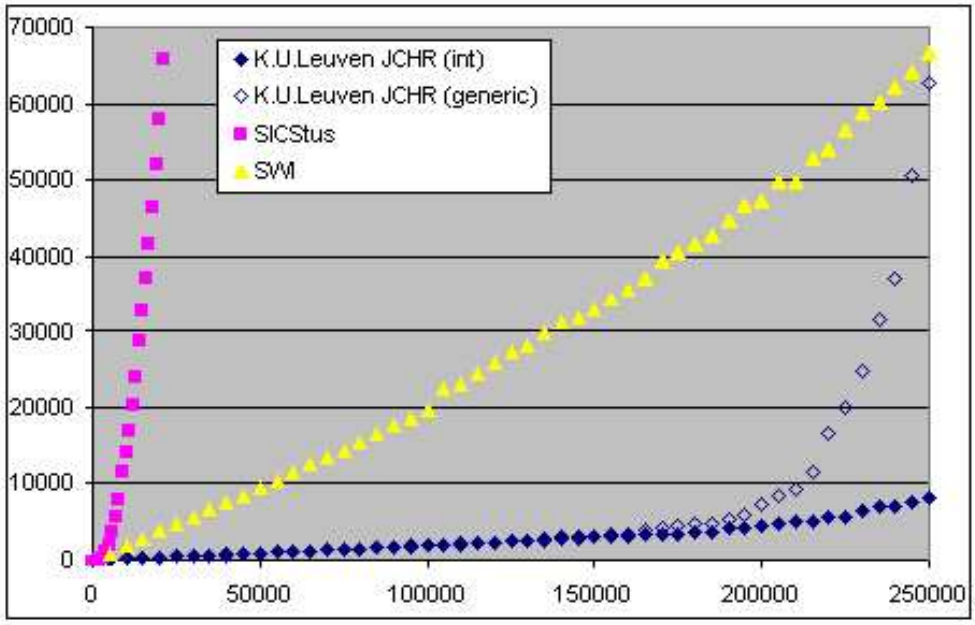
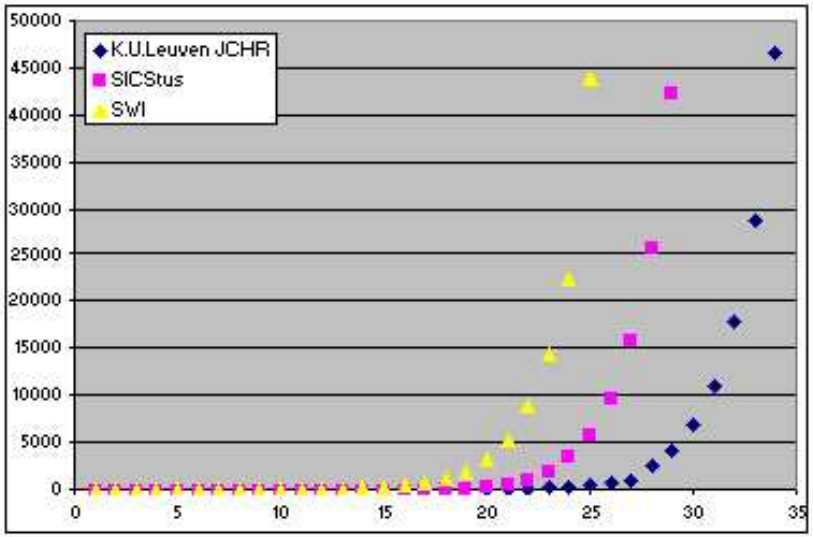
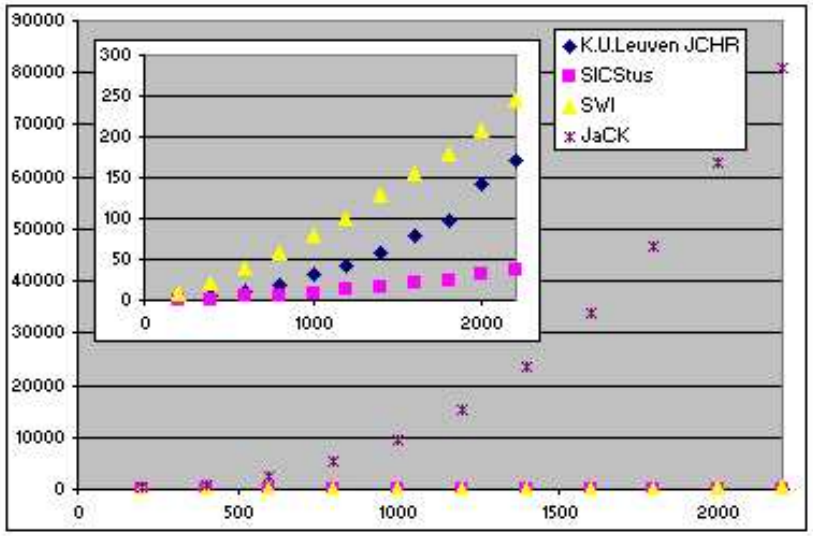


Fig. 5. Results for the union-benchmark.

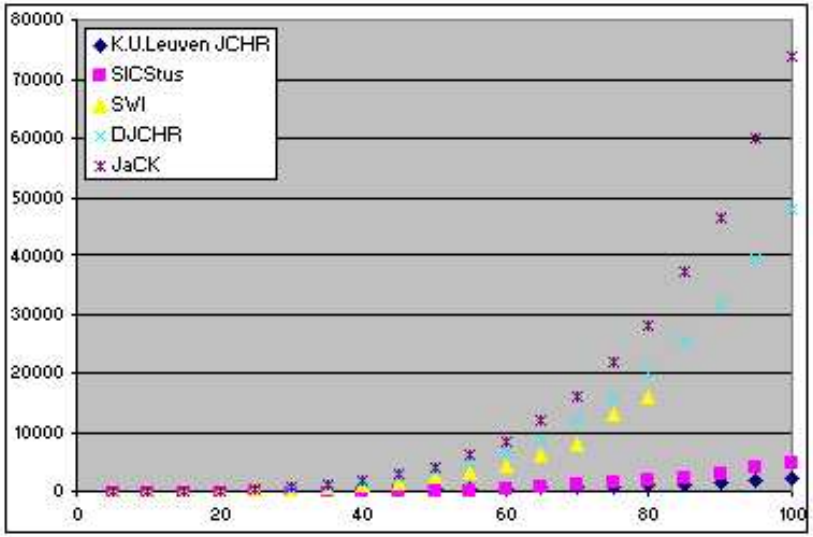
Figure 5 shows the results of the union-benchmark. **SWI** and **K.U.Leuven JCHR**, both featuring a constraint store indexed with a hash-table, show an optimal, quasi-linear behavior. **SICStus** on the other hand, lacking the necessary constant time operations on the constraint store, has a complexity that appears to be quadratic or worse. For **K.U.Leuven JCHR** the figure contains benchmark-results for both a version of the union-find handler using primitive integers and a generic version, using String objects to be precise. The performance difference between the two is negligible, until the latter causes garbage collection to become a performance-bottleneck.



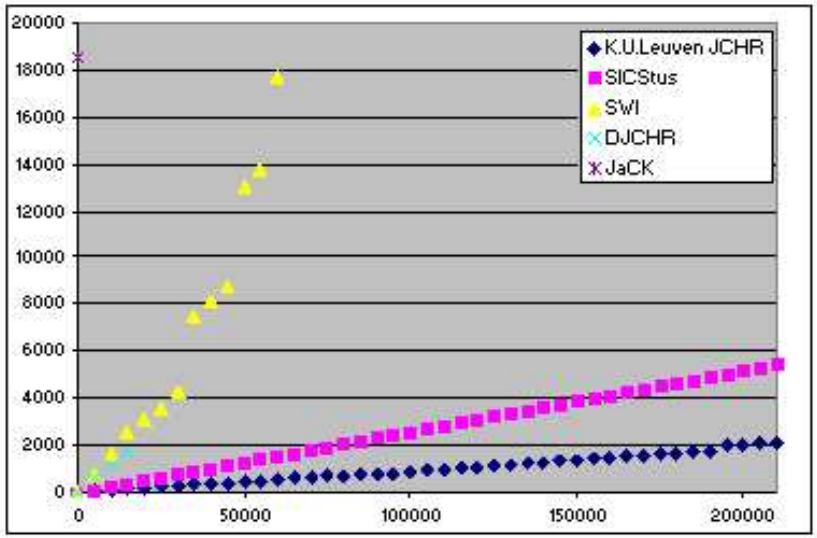
(a) fib.png



(b) primes.png



(c) leq.png



(d) bool.png

Fig. 6. Benchmark results.

In the Fibonacci-benchmark (Fig. 6(a)) the three systems considered behave similarly. The same holds for this trio in the next benchmark, performing the well-known "Sieve of Eratosthenes" algorithm (Fig. 6(b)). Furthermore this benchmark also clearly illustrates the lack of efficiency in **JaCK**. While the other systems remain within a quarter of a second, **JaCK** rapidly takes more than a minute and a half. This performance gap is also noticeable in the two remaining benchmarks: the leq- (Fig. 6(c)) and bool-benchmark (Fig. 6(d): see the top-left corner!).

Memory seems the only limiting factor in most benchmarks, with the apparent exception of the fib-benchmark[4]. In all but this last benchmark and the highly recursive primes-benchmark (where it is a stack-overflow that occurs) **K.U.Leuven JCHR** eventually runs out of heap-space. The same occurs to **DJCHR** sometimes even a lot sooner, like e.g. in the bool-benchmark. Memory-difficulties are not restricted to the Java systems: **SWI** too eventually overflows a stack in the fib-, leq- and bool-benchmark. In this respect **SICStus** clearly outperforms the other systems.

Time-wise, **K.U.Leuven JCHR** outperforms most often the other systems. There are several diverse reasons for that (cf. also Section 3.3), but some general observations can be made. We have already noted that the performance gap with **JaCK** can largely be explained by the fact that **JaCK** does not really compile CHR. The difference with **DJCHR** might be partially caused by the extent of book-keeping needed for the dynamic requirements. In fairness **K.U.Leuven JCHR** also does not yet have to keep the book-keeping needed for backtracking, but we are confident this can be added efficiently. An important factor contributing to the performance of **K.U.Leuven JCHR** is undoubtedly the just-in-time compilation in a mature Java-runtime, which can account for up to a an order of magnitude.

5 Conclusion

Our contribution is the K.U.Leuven JCHR system, a user-friendly, flexible and efficient CHR system in Java. The proposed high-level, declarative syntax and the full compliance to the refined operational semantics offer an intuitive look and feel, and easy porting of already existing CHR code. The flexible design allows for an integration of arbitrary static program analyses, compilation optimizations, hash-indices and constraint solvers, as demonstrated by features already implemented. We have shown how to apply the optimized compilation techniques of CHR for Prolog to the object-oriented programming paradigm and that this leads to superior efficiency.

The K.U.Leuven JCHR system is available at `http://www.cs.kuleuven.be/~toms/CHR/JCHR/`.

[4] However when one uses a simpagation rule instead of a simplification rule for the first rule (cf. the website for the program), memory once again becomes the only limiting factor instead of the execution-time!

5.1 Future Work

We intend to add support for backtracking to the system. This feature is essential when CHR is to be used for partial constraint solvers that require some form of search. For backtracking the constraint store data structures need to be extended with support for undoing updates. However, the most important challenge is to investigate how ordinary Java method calls should interact with backtracking. Useful ideas could come from JaCK's generic search engine JASE.

An efficient implementation of CHR in Java opens the door for efficient CHR implementations in other imperative (object-oriented) languages, such as C, C#, Visual Basic, ... The general principles are already outlined in this paper, but the tight integration of CHR with the particular host language should be considered for each language separately.

References

1. The FreeMarker website. `http://www.freemarker.org`.
2. Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In Gert Smolka, editor, *Proceedings of Third International Conference on Principles and Practice of Constraint Programming*, pages 252–266, Schloss Hagenberg, Austria, 1997. Springer LNCS.
3. Slim Abdennadher. *Rule-based Constraint Programming: Theory and Practice*. PhD thesis, Institute of Computer Science, LMU, Munich, 2001.
4. Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauss. JaCK: A Java Constraint Kit. In *Proceedings International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001)*, Kiel, Germany, September 13-15 2001. Society of Logic Programming, University of Kiel; published as Tech Report No. 2017.
5. Slim Abdennadher and Matthias Saft. A Visualization Tool for Constraint Handling Rules. In *Proceedings of 11th Workshop on Logic Programming Environments*, 1th December 2001.
6. Gilad Bracha. *Generics in the Java Programming Language*. Sun Microsystems, Inc., July 2004.
7. Manuel Carro et al. An introductory course on Constraint Logic Programming. `http://clip.dia.fi.upm.es/~vocal/public_info/`, December 1998.
8. H. Chun. Constraint Programming in Java with JSolver. In *Proceedings of the First International Conference and Exhibition on the Practical Application of Constraint Technologies and Logic Programming*, London, 1999.
9. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. Extending arbitrary solvers with Constraint Handling Rules. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 79–90, New York, NY, USA, 2003. ACM Press.
10. 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*, pages 90–104, Saint-Malo, France, September 2004.

11. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriott, Eds.), Journal of Logic Programming*, 37(1–3):95–138, October 1998.
12. Thom Frühwirth, Alexander Herold, Volker Küchenhoff, Thierry Le Provost, Pierre Lim, Eric Monfroy, and Mark Wallace. Constraint Logic Programming – an informal introduction. Technical Report ECRC-93-5, ECRC, Munich, Germany, February 1993.
13. Zvi Galil and Giuseppe F. Italiano. Data Structures and Algorithms for Disjoint Set Union Problems. In *ACM Computing Surveys (CSUR)*, volume 23, pages 319–344. September 1991.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
15. Christian Holzbaur, María García de la Banda, David Jeffery, and Peter J. Stuckey. Optimizing compilation of constraint handling rules. In *Proceedings of the 17th International Conference on Logic Programming*, pages 74–89, London, UK, 2001. Springer-Verlag.
16. Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules compiler and runtime system. *Applied Artificial Intelligence*, 14(4):369–388, April 2000.
17. Ekkehard Krämer. A generic search engine for a Java Constraint Kit. Master's thesis, Institute of Computer Science, LMU, Munich, Germany, January 15 2001.
18. Terence J. Parr et al. The ANTLR website. `http://www.antlr.org`.
19. Matthias Schmauß. An Implementation of CHR in Java. Master's thesis, Department of Computer Science, University of Munich, Germany, 1999.
20. Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium, jun 2005.
21. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In Thom Frühwirth and Marc Meister, editors, *Proceedings of the First Workshop on Constraint Handling Rules: Selected Contributions*, Ulm, Germany, May 2004.
22. Tom Schrijvers and Thom Frühwirth. Implementing and analysing union-find in CHR. Report CW 389, K.U.Leuven, Department of Computer Science, July 2004.
23. Mark E. Stickel. The path-indexing method for indexing terms. Technical Report 473, 333 Ravenswood Ave., Menlo Park, CA 94025, 1989.
24. Naoyuki Tamura. Cream: Class library for constraint programming in java. `http://bach.istc.kobe-u.ac.jp/cream/`, 2004–?
25. Paolo Viappiani et al. JCL – the Java Constraints Library. `http://liawww.epfl.ch/JCL/`.
26. Peter Van Weert. Constraint Programming in Java: een gebruiksvriendelijk, flexibel en efficiënt CHR-Systeem voor Java. Master's thesis, K.U.Leuven, May 2005.
27. Armin Wolf. Adaptive constraint handling with CHR in Java. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, number 2239 in Lecture Notes in Computer Science, Paphos, Cyprus, Nov./Dec. 2001. Springer-Verlag.
28. Armin Wolf, Thomas Gruenhagen, and Ulrich Geske. On incremental adaptation of CHR derivations. *Journal of Applied Artificial Intelligence*, Special issue on Constraint Handling Rules, 2000.

Constraint Abduction and Constraint Handling Rules

Martin Sulzmann[1], Jeremy Wazny[2] and Peter J. Stuckey[2]

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

Abstract. Type inference for advanced type extensions demands constraints consisting of implications, universal quantifiers etc. Hence, the current CHR solving which rewrites sets of primitive constraints is not applicable. We define an extension of the CHR solving mechanism which incorporates abductive reasoning to solve implication constraints. We state some fundamental results such as soundness and under some sufficient conditions completeness and termination. We have implemented a prototype solver as part of the Chameleon system. Our results allows us to advance the state of art in constraint solving and achieve inference for expressive type extensions.

1 Introduction

In our previous work [SS02,GJDS04], we have used Constraint Handling Rules[Frü95] (CHRs) as a powerful formalism to specify type inference systems. The benefits of CHRs should be clear. The standard approach towards type inference [Wan87,Mil78] generates a set of constraints out of the program text which are then solved by some domain-specific solver. Such solvers can often naturally be expressed in terms of CHRs [CF02,GJDS04]. In particular, we could establish some general criteria in terms of CHRs under which type inference via CHR solving is sound, complete and decidable [SS02].

However, there are a number of typing features whose type inference demands implication constraints rather than sets of constraints to capture precisely the typing problem. Typical examples are type annotations and pattern matching over constrained constructors of data types. By implication constraints we refer to constraints which are composed of Boolean conjunction, implication and a mix of universal and existential quantifiers. We give a brief overview in Section 3.

To actually obtain a type inference result, we yet need to find a solution to these implication constraints. Obviously, we could take the implication constraint itself as the solution. But then the problem of consistency checking becomes theorem proving. Something we definitely want to avoid. After all, type inference

should remain (fairly) simple. Another problem is that types become unreadable. Users of typed programming languages are accustomed to type descriptions in the form of sets of constraints but will be overwhelmed by more complex forms such as implication constraints.

In its simplest form, the problem we are facing is as follows. Given $C_1 \supset C_2$ find C_0 such that $C_0 \wedge C_1 \supset C_2$. This is known as abduction [Pei65,Mah05]. We concentrate on the framework of *constraint abduction* [Mah05], where we have no notion of abducible predicates that would limit the answers to C_0 we will allow. In particular we are interested in abducing conjunctions of Herbrand equations and predicates (or in our terminology user-defined constraints) that force the implication, rather than simply abducible predicates, and we have to contend with nested quantifiers in the formula C_2.

In our setting, the meaning of user-defined constraints is defined in terms of CHRs. Hence, we need to find C_0 such that $P \models C_0 \wedge C_1 \supset C_2$ where P refers to the logical reading of CHRs.

In this paper, we consider an extension of the CHR solving mechanism to allow for abductive reasoning. Other work on abduction and CHRs e.g. [SAC00] concentrates on using CHRs to specify and solve the standard abduction problem, C_1 is a logic program with integrity constraints, and C_0 are limited to be a set of abducible predicates. Here we extend CHRs to allow implication constraints to be part of the language, and extend the CHR solving approach to tackle these constraints. This requires us to solve constraint abduction problems. To the best of our knowledge, such an extension has not been considered before.

Specifically, our contributions are:

- We extend the CHR solving mechanism with abductive reasoning to solve implication constraints (Section 4.1).
- We establish some soundness, completeness and termination results (Section 4.3).
- We report our experiences with an implementation of an instance of our implication solving scheme (Section 5).

In Section 2, we provide some background material about CHRs and give a brief overview of our previous work on CHR type applications. We conclude in Section 6. The curious reader may take a look at Section 4 for a quick overview of our solving method for implications constraints.

Proofs and background material on the precise connection between type inference and implication constraints can be found here [SWS05].

2 Background

We first review the semantics of CHRs (Section 2.1). Then, we take a look at our previous work where we have used CHRs to model type relations (Section 2.2).

2.1 Constraint Handling Rules

The syntax of CHRs is as follows.

$$
\begin{array}{lll}
\text{Terms} & t & ::= v \mid a \mid t \rightarrow t \mid (t,t) \mid ... \\
\text{Constraint} & C & ::= t = t \mid U\ t \mid C \wedge C \\
\text{Simplification} & s & ::= c_1, ..., c_n \Longleftrightarrow d_1, ..., d_m \\
\text{Propagation} & p & ::= c_1, ..., c_n \Longrightarrow d_1, ..., d_m \\
\text{CHR} & P & ::= s \mid p \mid P, P
\end{array}
$$

We assume a standard Herbrand term language consisting of variables v, constants a, function constructors $(\cdot \rightarrow \cdot)$, pairs $((\cdot,\cdot))$, and other constructors etc. Constraints consist of equations among Herbrand terms $(t = t)$ and user-defined constraints $(U\ t)$ to which we sometimes refer to as *primitive* constraints. Note that we treat constraints as sets. Hence, we often abbreviate $\wedge$ by "," and use set notation for a sequence of primitive constraints. Note that we make use of this Haskell like notation for CHRs, rather than the usual Prolog like notation (although we show correspondences for a few examples).

We make use of CHR *simplification* and *propagation* rules. We assume that c_i refer to user-defined constraints and d_i refer to user-defined constraints and equations. Sometimes, we refer to a set P of CHRs as a CHR program.

We use $fv(F)$ to denote the free variables in formula F. We use $\bar{\exists}_V F$ to denote the existential quantification of variables in F other than varialbes in the set V, i.e. $\exists x_1 \cdots \exists x_n F$ where $\{x_1, \ldots x_n\} = fv(F) - V$.

The logical reading of CHRs is straightforward. Details can be found in [Frü95].

The operational reading of CHRs is as follow. In case of propagation rules we add the right hand side if we find a matching copy of the left hand side in the constraint store. In case of simplifications rules we remove the matching copy and replace it by the right hand side. The formal definition is as follows.

Definition 1 (CHR Semantics). *Let P be a set of CHRs. Let $(R)\ \ c_1, ..., c_n \Longrightarrow d_1, ..., d_m \in P_p$ and C be a constraint. Let ϕ be an m.g.u. of all equations in C. Let $c'_1, ..., c'_n \in C$ such that there exists a substitution θ on variables in rule (R) such that $\theta(c_i) = \phi(c'_i)$ for $i = 1...n$, that is user-defined constraints c'_1,...,c'_n* match *the left-hand side of rule (R). Then, $C \rightarrowtail_R C, \theta(d_1), ..., \theta(d_m)$.*

Let $(R)\ \ c_1, ..., c_n \Longleftrightarrow d_1, ..., d_m \in P_p$ and C be a constraint. Let ϕ be the m.g.u. of all equations in C. Let $c'_1, ..., c'_n \in C$ such that there exists a substitution θ on variables in rule (R) such that $\theta(c_i) = \phi(c'_i)$ for $i = 1, ..., n$. , Then, $C \rightarrowtail_R C - \{c'_1, ..., c'_n\}, \theta(d_1), ..., \theta(d_m)$.

Often, we perform some equivalence transformations (e.g. normalize equations by building the m.g.u. etc) which are either implicit or explicitly denoted by $\longleftrightarrow$. A *derivation*, denoted $C \rightarrowtail^*_P C'$ is a sequence of derivation steps using rules in P such that no further derivation step is applicable to C'. CHRs are applied exhaustively, being careful not to apply propagation rules twice on the same constraints (to avoid infinite propagation). For more details on avoiding re-propagation see e.g. [Abd97]. We say a set of CHRs is *terminating* if for each C there exists C' such that $C \rightarrowtail^*_P C'$. The CHR Soundness Result [Frü95] guarantees that if $C \rightarrowtail^*_P C'$ then $P \models C \leftrightarrow \bar{\exists}_{fv(C)}.C'$. This shows that CHR rule applications perform equivalence transformations.

Note that in our description of the CHR semantics we assume set semantics. This is in contrast to the standard description which assumes a multi-set semantics [Frü95]. Set semantics is vital for our purposes. Thus, we are able to build first-order models which allows us to state some strong results. E.g., under some sufficient conditions we can guarantee canonical normal forms [SS02].

2.2 Type Inference via CHRs

Our main interest in CHRs lies in modeling type relations. A typical example are type classes which allow to specify a user-definable constraint domain.

Example 1. Here is a Haskell type class program making use of multi-parameter type classes [JJM97] with functional dependencies [Jon00].

```
data Zero
data Succ x
class Add l m n | l m -> n
instance Add Zero m m            -- (A1)
instance Add l m n =>
         Add (Succ l) m (Succ n) -- (A2)
```

We introduce a constant type constructor `Zero` and a unary type constructor `Succ` to represent natural numbers. The functional dependency `| l m -> n` imposed on the ternary type class `Add` states that the first two parameters uniquely determine the third one. The instances provide for an inductive definition of addition among our representation of natural numbers. Their meaning will become clear shortly when translating type classes to CHRs. Note that type classes usually come with a set of methods and instance bodies. We omit these details here for simplicity.

Type classes follow the open world assumption, meaning a new instance could be available at any time. Hence, it is very natural to encode them in terms of CHRs.

Example 2. Here are the CHRs resulting from the above example. Note that our CHR syntax only allows for unary user-defined constraints. Hence, we silently consider $Add\ a\ b\ c$ as a short-hand for $Add\ (a, b, c)$.

$$
\begin{array}{ll}
\text{(A1)} & Add\ Zero\ m\ m \iff True \\
\text{(A2)} & Add\ (Succ\ l)\ m\ (Succ\ n) \iff Add\ l\ m\ n \\
\text{(A3)} & Add\ l\ m\ n_1, Add\ l\ m\ n_2 \implies n_1 = n_2 \\
\text{(A4)} & Add\ Zero\ m\ n \implies m = n \\
\text{(A5)} & Add\ (Succ\ l)\ m\ n' \implies n' = Succ\ n
\end{array}
$$

Note that we use "type class" syntax to write CHRs. In a more usual CHR syntax we have

```
A1 @ add(0,M,N)              <=> true.
A2 @ add(s(L), M, s(N))      <=> add(L,M,N).
```

```
A3 @ add(L,M,N1), add(L,M,N2) ==> N1 = N2.
A4 @ add(0,M,N)               ==> N = M.
A5 @ add(s(L),M,N)            ==> N = s(_).
```

The important point is that each instance declaration is turned into a simplification rule, see rules (A1-2). The functional dependency gives rise to rule (A3). Rules (A4-5) enforce the functional dependency for each specific instance. We refer the interested reader to [SS02,GJDS04] for more details how to translate type classes to CHRs.

Thus, we can explain the type class semantics in terms of CHR solving. For example, assume some program text generates the constraint

$$Add\ (Succ\ Zero)\ m\ n, Add\ (Succ\ Zero)\ m\ n', m = Zero$$

Type inference becomes CHR solving. We have that

$$\begin{array}{ll} & Add\ (Succ\ Zero)\ m\ n, Add\ (Succ\ Zero)\ m\ n', m = Zero \\ \rightarrowtail_{A3} & Add\ (Succ\ Zero)\ m\ n, m = Zero, n = n' \\ \longleftrightarrow & Add\ (Succ\ Zero)\ Zero\ n, m = Zero, n = n' \\ \rightarrowtail_{A5} & Add\ (Succ\ Zero)\ Zero\ (Succ\ n''), n = Succ\ n'', m = Zero, n = n' \\ \rightarrowtail_{A1} & Add\ Zero\ Zero\ n'', n = Succ\ n'', m = Zero, n = n' \\ \rightarrowtail_{A4} & Add\ Zero\ Zero\ Zero, n'' = Zero, n = Succ\ Zero, m = Zero, n = n' \\ \rightarrowtail_{A2} & n'' = Zero, n = Succ\ Zero, m = Zero, n = n' \end{array}$$

Note that we apply set semantics (see the first CHR derivation step). In the third and fifth step, we perform some implicit equivalence transformation steps. The result is a type constraint that gives the types for the variable n, n', m which correspond to program fragments.

Here is the above derivation written in the more usual CHR syntax.

```
     add(s(0), M, N1), add(s(0), M, N2), M = 0
↣A3  add(s(0), M, N2), M = 0, N1 = N2
⟷    add(s(0), 0, N2), M = 0, N1 = N2
↣A5  add(s(0), 0, s(N3)), N2 = s(N3), M = 0, N1 = N2
↣A1  add(0, 0, N3), N2 = s(N3), M = 0, N1 = N2
↣A4  add(0, 0, 0), N3 = 0, N2 = s(N3), M = 0, N1 = N2
↣A2  N3 = 0, N2 = s(N3), M = 0, N1 = N2
```

For the rest of the paper, we will however use the "type class" syntax of CHRs.

3 Advanced Typing via Implication Constraints

We briefly motivate the need for implication constraints whose syntax is as follow.

$$\text{Implication Constraints } F ::= C \mid \forall \bar{a}.(C \supset \exists \bar{b}.F) \mid F \wedge F$$

Note that we consider a restricted set of first-order formula. E.g., assumptions (left-hand side of $\supset$) only consist of sets of primitive constraints. In a single implication constraint $\forall \bar{a}.(C \supset \exists \bar{b}.F)$ we find universal and existential quantifiers. Note that F may contain again a mix of universal and existential quantifiers. Hence, we can immediately encode the unification problem under a mixed prefix [Mil92] (if we take all assumptions to be $True$ and equations are the only primitive constraint).

For our purposes, implication constraints have a lot of other uses. Implication constraints allow us to capture precisely the typing problem for some typing features found in Haskell [Has] with extensions. Note that similar arguments can be made for other languages such as Mercury [Hen01], HAL [DGH+99] etc. For concreteness, we use Haskell syntax in the following.

Example 3. Here is a (contrived) program making use of type annotations.

```
class Foo a b where foo::a->b->Int
instance Foo a Int  where foo _ _ = 1    -- (F1)
instance Foo a Bool where foo _ _ = 2    -- (F2)
f x = (x::Int, let g :: a->Int
                   g y = foo y x
               in g x)
```

Note that variable `a` in `g`'s annotation is (implicitly) universally quantified. Hence, we need to check whether `g`'s program text satisfies the annotation. The following implication constraint represents the check for correctness of `g`'s type annotation.

$$\forall a.(t_g = a \to Int \supset \exists t_y.(t_g = t'_g, t'_g = t_y \to Int, Foo\ t_y\ t_x))$$

The above constraint holds in combination with $t_x = Int$ (generated from the surrounding context), $t_f = t_x \to (t_x, Int)$ and the instance (F). Hence, the expected type of `f` is `Int->(Int,Int)`.

Note that without implication constraints we would be forced to perform the annotation check locally. Hence, we might miss the information that $t_x = Int$. Hence, the annotation check fails and this is indeed the case for the standard Haskell implementations [GHC,HUG].

In the next example, we constrain the possible types of a data type constructor [Läu96].

Example 4. Consider

```
class Key a where getKey :: a->Int
instance Key Bool where
  getKey True = 1
  getKey False = 0
instance Key Char where
  getKey = charToInt
data KEY = forall a. Key a => Mk a
h (Mk x) = getKey x
```

The `forall a` keyword indicates that the argument type of constructor `Mk` does not appear in the type of the data type `KEY`. Hence, values of type `a` are "abstract". Keyword `Key a =>` constrains values of type `a`. This implies that when accessing a `KEY` value we may make use of the `Key` class and its associated methods. Consider function `f` which pattern matches over `KEY`. The implication constraint representing the typing problem here is as follows.

$$t_f = KEY \rightarrow t_1, \forall a.(Key\ a \supset (Key\ a, t_1 = Int))$$

Thus, we are able to represent the temporary assumption $Key\ a$ when typing the function body. The universal quantifier $\forall a$ ensures that the `KEY` values remains abstract. We can verify that $t_f = KEY \rightarrow Int$ solves the above constraint. Hence, `h` can be given type `KEY -> Int`.

4 Solving of Implication Constraints

We consider the task of solving an implication constraint F whose meaning of user-defined constraints is defined in terms of some CHR program P. We are interested in finding a constraint C (consisting of sets of primitive constraints) such that $P \models C \supset F$. Commonly, we refer to C as a *solution* of F w.r.t. P. Note that $False$ is a trivial solution which could obviously be excluded. In the following, we lift the CHR solving mechanism to the domain of implication constraints. We write $F \gg^*_P C$ to denote that solving of F yields solution C.

Our idea is to turn the implication constraint solving problem into an equivalence checking problem by making use of the fact that $C_1 \supset C_2$ iff $C_1 \leftrightarrow C_1, C_2$. Then, we execute $C_1 \rightarrowtail^*_P D_1$ and $C_1, C_2 \rightarrowtail^*_P D_2$ and check whether D_1 and D_2 are logically equivalent. If yes, then $C_1 \supset C_2$ is solved and $True$ is the solution. Hence, $C_1 \supset C_2 \gg True$. Otherwise, we observe final stores to guess a (partial) solution S. Hence, $C_1 \supset C_2 \gg S, (C_1 \supset C_2)$

The obvious guess is to take $S = D_2 - D_1$, i.e. we take the difference between both final stores. However, we allow for universal quantifications. E.g., consider $\forall a.(Bar\ a \supset Foo\ a)$. Clearly, $Foo\ a$ is not a valid solution (assuming no CHRs are imposed) because the quantified variable a escapes. Hence, we need to exclude partial solutions which are bound by some universally quantified variables.

We also allow for nested implications. E.g., consider $\forall a.(Foo\ a \supset \forall b.(Bar\ a\ b \supset ...))$. We treat such cases by recursive solving. In case, we encounter a sequence of implication constraints. We pick one single implication constraint at a time.

4.1 CHR Implication Solver with Abductive Reasoning

The description of our generic implication solving scheme is as follow.

Definition 2 (CHR Implication Solver). *Let P be a set of CHRs and F an implication constraint.*

Primitive: *We define $F \gg_P C'$ where $C \rightarrowtail^*_P C'$ if $F \equiv \exists \bar{a}.C$.*

General: *Otherwise* $F \equiv C_0, (\forall \bar{a}.D \supset \exists \bar{b}.F_1), F_2$ *where* C_0 *is a conjunction of primitive constraints,* D *is a set of assumptions and* F_1 *and* F_2 *are implication constraints.*
We compute (1) $C_0, D \rightarrowtail_P^* D'$ *and (2)* $C_0, D, F_1 \gg_P^* C'$ *for some* D' *and* C'*. We distinguish among the following cases.*
Solved: *We define* $F \gg_P C_0, F_2$ *if* $\models (\bar{\exists}_V.D') \leftrightarrow (\bar{\exists}_V.C')$ *where* $V = fv(C_0, D, \bar{a})$*.*
Add: *We define* $F \gg_P S, F$ *if* $\models (\bar{\exists}_V.D') \not\leftrightarrow (\bar{\exists}_V.C')$ *where* $V = fv(C_0, D, \bar{a})$ *and* $False \notin C'$ *and* S *is a conjunction of primitive constraints such that (3)* $P \models C' \supset S$*, (4)* $fv(S) \cap \bar{a} = \emptyset$ *and (5)* $P \models D' \not\supset S$*.*
Failure: *We define* $F \gg_P False$ *in all other cases.*

We assume that $\gg^*$ *denotes the exhaustive application of CHR implication solving steps.*

In the **Primitive** step we apply standard CHR solving. No surprises here. In the **General** step, we split the constraint store into C_0 containing sets of primitive constraints, a single implication constraint $(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$ and F_2 containing the remaining implication constraints. Strictly speaking, F_2 itself could be a set of primitive constraints. Silently, we assume that all sets of primitive constraints are collected in C_0. The important point to note here is that we do not specify an order in which to solve constraints. We come back to this issue shortly. Also note that we inductively solve nested implication constraints, see (2).

The **Solved** step applies if the equivalence check succeeds. Hence, the constraint $(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$ is removed from the store. Note that w.l.o.g. we assume that $\bar{b} = fv(F_2) - fv(\bar{a}, C_0)$. Any variable not bound by a universal quantifier is (implicitly) existentially bound.

The **Add** step is the most interesting solving step and applies if equivalence checking failed. We perform some abductive reasoning to find a constraint S which may help us to eventually solve the implication. Often, we refer to S as a *partial solution*. Conditions (3-5) give only a characterization which subset of primitive constraints in the final store of derivation (2) may prove useful to solve the implication constraint $(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$. The least we require is the following. Condition (3) ensures that we only add in useful constraints. Condition (4) prevents universally bound variables $\bar{a}$ from escaping. Condition (5) avoids adding in redundant constraints.

4.2 Examples

We consider a couple of examples to get a feel for CHR implication solving.

Example 5. Consider $P = \{(\mathrm{F}) \quad Foo\ Int\ b \Longleftrightarrow True\}$ and

$$F = \left\{ \begin{array}{ll} t = Erk\ a \rightarrow t_3, t_3 = t_1, t_3 = t_2, & (C_0) \\ \forall b.(Bar\ a\ b \supset Bar\ a\ b, t_1 = a), & (\forall b.D_1 \supset C_1) \\ \forall b.(True \supset Foo\ a\ b, t_2 = Int) & (\forall b.D_2 \supset C_2) \end{array} \right\}$$

Solving proceeds as follow:
Step 1: Solve $C_0, (\forall b.D_1 \supset C_1)$: Add $t_1 = a$. Hence, $F \gg F, t_1 = a$.
Step 2: Solve $C_0, t_1 = a, (\forall b.D_2 \supset C_2)$: Add $t_2 = Int$. Hence, $F, t_1 = a \gg F, t_1 = a, t_2 = Int$.
Step 3: Solve $C_0, t_1 = a, t_2 = Int, (\forall b.D_1 \supset C_1), (\forall b.D_2 \supset C_2)$: Solved. Hence, $F, t_1 = a, t_2 = Int \gg t = Erk\ a \to t_3, t_3 = t_1, t_3 = t_2, t_1 = a, t_2 = Int$.

In the above example, we have been solving implications in turn, combining results until we have solved all implications or reached failure. In the following, we employ a different solving strategy.

Example 6. We assume the definitions from Example 5. We consider solving of each implication individually. That is, we perform solving until we have solved the implication or reached failure. We do not combine results here.
Step 1: Solve $C_0, (\forall b.D_2 \supset C_2)$: Add $t_2 = Int$. Hence, $F \gg F, t_2 = Int$.
Step 2: Solve $C_0, t_2 = Int, (\forall b.D_2 \supset C_2)$:

$$\begin{array}{ll} C_0, t_2 = Int, D_2 & \rightarrowtail^* t = Erk\ a \to t_3, t_3 = t_1, t_3 = t_2, t_2 = Int \\ C_0, t_2 = Int, D_2, C_2 & \rightarrowtail^* t = Erk\ a \to t_3, t_3 = t_1, t_3 = t_2, t_2 = Int, \\ & \qquad Foo\ a\ b, t_2 = Int \end{array}$$

As Example 5 shows we could solve the implication constraint by adding $t_1 = a$. Clearly, we need some reasonably sophisticated reasoning about the set of CHRs P and the constraint store to find that $t_1 = a$ is a "good" guess. A naive implementation of the add step will be unable to find any constraint which could be added. Hence, we fail.

In general, the best solving strategy seems to iterate over the set of implications. However, this may lead to a high solving cost (in terms of the underlying CHR solver).

Example 7. Consider $P = \{$(F) $F\ Int\ Int\ b \iff True$ ❙ (G1) $G\ Int\ Int\ b \iff True$ ❙ (G2) $G\ Int\ c\ b \implies c = Int\}$ and

$$F = \left\{ \begin{array}{ll} t = Erk\ a_1\ a_2 \to (Int, Int), & (C_0) \\ \forall b.(True \supset F\ a_1\ a_2\ b, a_1 = Int), & (F_1) \\ \forall b.(True \supset G\ a_1\ a_2\ b) & (F_2) \end{array} \right\}$$

Solving by iterating over the set of implications proceeds as follows.
Step 1: Solve C_0, F_1: Add $a_1 = Int$. Hence, $F \gg F, a_1 = Int$.
Step 2: Solve $C_0, a_1 = Int, F_2$: Add $a_2 = Int$ (rule (G2) applies). Hence, $F, a_1 = Int \gg F, a_1 = Int, a_2 = Int$.
Step 3: Solve $C_0, a_1 = Int, a_2 = Int, F_1$: Solved! (rule (F) applies). Hence, $F, a_1 = Int, a_2 = Int \gg C_0, a_1 = Int, a_2 = Int, F_2$.
Step 4: Solve $C_0, a_1 = Int, a_2 = Int, F_2$: Solved! (rule (1) applies). Hence, $C_0, a_1 = Int, a_2 = Int, F_2 \gg C_0, a_1 = Int, a_2 = Int$.

Notice how the sub-results affect each other. Each time we add some partial solution for one implication, we achieve progress (i.e. add a new partial solution)

for the other implication. Obviously, we could generalize this example such that we need to jump back and forth between implications almost an arbitrary number of times.

Another issue is that the order of solving may affect the final result.

Example 8. Consider $P = \{Foo\ b\ Int \Longrightarrow Bar\ b\}$ and

$$(Foo\ b\ a \supset Bar\ b), (Foo\ b\ a \supset a = Int)$$

Solving the second implication constraint first yields the final result $a = Int$. In case, we switch the order we first find the partial solution $Bar\ b$ which then yields the final result $Bar\ b, a = Int$.

4.3 Results

We can state the following result for any concrete solver instance satisfying Definition 2.

Lemma 1 (Soundness). *Let P be a set of CHRs, F be an implication constraint and C be a set primitive constraints such that $F \gg_P^* C$. Then, $P \models C \supset F$.*

The above result is important to guarantee that type inference is sound. Less obvious is how to achieve "best" solutions.

Definition 3 (Principal Solutions). *Let P be a set of CHRs, F be an implication constraint. We say that C is a* solution *of F w.r.t. P iff $P \models C \supset F$.*

We say that C is a principal *solution iff C is a solution and for any other solution C' we have that $P \models C' \supset C$.*

We say that P satisfies the principal implication constraint property *iff each implication constraint has a principal solution or no solution exists.*

Immediately, we find that the principal implication constraint property cannot be satisfied in general.

Example 9. Consider $P = \{Foo\ a\ Int \Longleftrightarrow True \mid Foo\ a\ Bool \Longleftrightarrow True\}$ and $\forall a.True \supset Foo\ a\ t_x$. We find that $t_x = Int$ and $t_x = Bool$ are two incomparable solutions and there is no more general solution. In fact, this shows that Haskell does not enjoy principal types. E.g., the following variation of Example 3

```
f' x = (x, let g :: a->Int
               g y = foo y x
           in g x)
```

can be given types `f'::Int->(Int,Int)` and `f'::Bool->(Bool,Int)` but there is no more general type.

For a restricted class of CHR programs and implication constraints however we can guarantee principal solutions.

We say a CHR program P belongs to the multi-parameter type class (*MPTC*) program class if the following three restrictions are satisfied.

1. CHRs are single-headed simplification rules where we only find user-defined constraints or $True$ on right-hand sides.
2. The rules are range-restricted. We say a CHR is *range-restricted* iff grounding all variables on the left-hand side of a CHR, grounds all variables on the right-hand side.
3. The CHRs are terminating and confluent. We say a set of CHRs is *confluent* iff any order of rule applications yields to the same final store.

We say a implication constraint F is *unambiguous* iff for any sub-formula $C_0, (\forall \bar{a}.D \supset \exists \bar{b}.F_1)$ encountered during solving we have that all variables in F_1 are either uniquely determined by C_0 and D or part of $\bar{a}$.

We say a implication constraint F is *pure* iff for each sub-formula $(\forall \bar{a}.D \supset \exists \bar{b}.F_1)$ we have that (a) D consists only of user-defined constraints, and (b) any user-defined constraint in F_1 consists of only universally bound variable.

MPTC programs in combination with the unambiguity condition ensure that the logical equivalence check in the **Solved** step holds iff the considered formula holds under the CHR program. The purity condition is important to ensure that we can locally compute principal solutions. E.g., consider Example 9 where condition (c) is violated.

Lemma 2 (Completeness). *Let P be a CHR program in MPTC. Then, P satisfies the principal implication constraint property for any unambiguous and pure implication constraint.*

Termination is another important property. Obviously, the CHR program must be terminating. However, this is not sufficient to guarantee termination. A subtle problem is that our current CHR implication solver description allows to add repeated, renamed copies of constraints. E.g., consider solving of $Bar\ a \supset Foo\ a$ where $P = \{Foo\ a \supset a = [b]\}$. We find that $Bar\ a \supset Foo\ a \gg a = [b_1], (Bar\ a \supset Foo\ a) \gg a = [b_1], a = [b_2], (Bar\ a \supset Foo\ a), ...$ is a valid derivation.

We can state termination under the condition that partial solutions are variable-restricted. Formally, partial solution S in Definition 2 is *variable-restricted* iff (6) $fv(S) \subseteq fv(C_0, D) - \bar{a}$. In essence, we prevent the addition of fresh variable.

Lemma 3 (Termination). *Let P be a terminating CHR program and F an implication constraint. We assume all partial solutions are variable-restricted. Then, $F \gg_P^* C$ for some C or we report failure.*

Unfortunately, the restriction to variable-restricted (partial) solutions is too restrictive in practice.

Example 10. Consider type inference for

```
data List n = (n=Zero) => Empty
            | forall m. (n=Succ m) => List Int m
f Empty = Empty
```

which gives rise to

$$t_f = List\ n \rightarrow t_1, (n = Zero \supset (t_1 = List\ m, m = Zero))$$

In an intermediate solving step, we find final stores

$$t_f = List\ n \rightarrow t_1, n = Zero$$

and

$$t_f = List\ n \rightarrow t_1, n = Zero, t_1 = List\ m, m = Zero$$

Taking $S = t_1 = List\ m, m = Zero$ leads to success but this partial solution is not variable-restricted.

Hence, we do not enforce variable-restriction in our actual implementation. An alternative termination condition may be the restriction to terminating and range-restricted CHR programs. However, also range-restriction is something which is often not satisfied by CHRs (see rule (A5) in Example 2). A rather simple method (that in practice seems to work well) is to impose a bound on the number of solving steps.

5 Implementation

We have implemented a prototype solver as part of the Chameleon system [SW]. Our generic implication solver is written in Haskell and built upon an optimized CHR solver [Duc]. It can handle not only stand-alone implication goals, but also implications which appear in the body of CHRs rules.

Our CHR solver uses an efficient, untyped term representation (WAM style), to encode both types and constraints. This allows for efficient implementations of term matching, unification, and equivalence checking. The fact that these algorithms directly update the terms they are passed, however, complicates the abduction step.

In solving an implication constraint under CHR rules P, $\forall \bar{a}.(D \supset \exists \bar{b}.C)$ in the context of primitives C_0, we perform two CHR derivations $C_0, D \rightarrowtail_P^* D'$, and $D', C \rightarrowtail_P^* C'$. If C' is equivalent to D', wrt variables $\bar{a}, fv(C_0, D')$, then we can eliminate this implication constraint, as it is trivially *True* in this context. Otherwise, we attempt to solve it, via abduction.

Performing abduction involves comparing the C' store against C_0, and extracting any additional information it contains about the variables $fv(C_0)$, which is not already part of D'. We then project C' onto $fv(C_0)$, essentially building a set of equations $S_{C'}$, which represent part of C_0 which was *potentially* updated in the step from D' to C'. We say that $S_{C'}$ represents only a possible update, because it may contain equations that merely rename the original C_0 variables, add therefore do not add any substantial information.

All that remains is to backtrack to the point before either derivation, and test whether adding $S_{C'}$ instantiates anything. If so, $S_{C'}$ represents a partial solution of the original implication. If not, then we could find no partial solution, and we fail.

A sketch of the implication solving algorithm can be found in Figure 1. We assume that solve is called whenever a normal CHR derivation, indicated by the usual arrow, $\rightarrowtail^*$ ends, with implications left over. It is invoked with two arguments, the first of which, F represents the implication and user constraints in the store, and the second P, the current CHR program. It returns any remaining user constraints, or implications. Herbrand constraints are treated implicitly, and updated directly as the result of unification. If F is primitive, we return it immediately, since there are no implications to process. Otherwise we split F into user constraints C_0 and implications Fs , and attempt to solve each implication in turn. We use variable V_{C_0} to record the variables in C_0 at that time, S to represent the current solution to the implications, and $F's$ to record the implications which have been only partially solved.

For each implication, we first set a choice point, and then perform the left and right-hand derivations. If the result of both derivations is equivalent, we can forget about this implication, and so we break out of the loop early. If either derivation failed, then we set S to *False* and also break out of the loop. Otherwise, we project the constraints in the current (otherwise implicit) Herbrand store onto the variables C_0, which gives us $S_{C'}$, our potential solution constraints. We then rewind, and check whether $S_{C'}$ adds anything to C_0. If not, the constraints we found represent no solution, so we set S to *False*, and break out of the loop. Otherwise, we add $S_{C'}$ to C_0, record the current implication in $F's$, to return to later, and continue to the next implication.

When we finally leave the loop, if S is *False*, then solving failed. If there remain no implications to solve in $F's$, then we can return the solution we have calculated, otherwise we recursively invoke solve with the partial solution we have so far, and all remaining implications. In practice, we may place a limit on the depth of this recursion, in order to guarantee termination.

Extending CHR solving to the domain of implication constraints allows to type check some interesting programs.

Example 11. Consider the following where we make use of the CHRs from Example 2.

```
data List a n = (n=Zero) => Nil
              | forall m. (n=Succ m) => Cons a (List a m)

append ::  Add l m n =>
           List a l -> List a m -> List a n
append Nil ys = Nil
append (Cons x xs) ys = Cons x (append xs ys)
```

The above is essentially the well-known append function. The difference is that we have refined the list data type which carries an additional type argument `n` describing the length of a specific list. The type of `append` carries the additional property that the sum of the length of the two input lists equals the length of the output list.

```
solve(F,P) =
    if primitive(F)
    then return F'
    else let (C_0, Fs) = F      (C_0 primitive, Fs nonprimitive)
         let V_{C_0} = fv(C_0)
         let S = True
         let F's = ∅
         for ∀ā.(D ⊃ ∃b̄.C) in Fs
             let cp_1 = set_choice_point()
             do C_0, D ↣*_P D'
             let V_{C_0,D'} = fv(D')
             do D', C ↣*_P C'
             let V = ā ∪ V_{C_0,D'}
             if C' = False then S = False, break
             if ⊨ (∃̄_V D') ↔ (∃̄_V C') then break
             let H = get_herbrand_store()
             S_{C'} := project(V_{C_0}, H)
             do rewind(cp_1)
             if ⊨ (∃̄_{V_{C_0}} C_0) ↔ (∃̄_{V_{C_0}} C_0, S_{C'}) then S = False, break
             S = S ∪ S_{C'}
             F's = F's ∪ {∀ā.(D ⊃ ∃b̄.C)}
         if S = False then return False
         if F's = ∅ then return C_0, S
         solve((C_0, S, F's), P)
```

Fig. 1. Implication solver with abduction

6 Conclusion

We have pointed out new directions in the applications of CHRs. We have presented an extension of CHRs to support type inference for advanced systems. Our solving approach relies on a combination of CHR solving, equivalence checking and abductive reasoning. We could show that our generic implication solver is sound and could state completeness for a restricted class of programs and constraints. We could also identify a termination condition by disallowing the addition of fresh variables. We have implemented our approach as part of the Chameleon system [SW].

Solving of implication constraints turned out to be a hard problem. There are cases where the solution we find is not principal, or we do not find a solution even if a solution exists (see Examples in Section 4.2 and Example 9). This is mainly due to the fact that local principal solutions may not exist. Hence, some more sophisticated abductive reasoning will be necessary which not only observes final stores but also the CHR program. This is a direction we plan to pursue in the future. We also intend to study the special case where the only primitive constraints are equations. Some recent work [Mah05] may prove to be useful here.

References

[Abd97] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proc. of CP'97*, LNCS, pages 252–266. Springer-Verlag, 1997.

[CF02] E. Coquery and F. Fages. TCLP: Overloading, subtyping and parametric polymorphism made practical for CLP. In *Proc. of ICLP '02*, volume 2401, page 480. Springer-Verlag, 2002.

[DGH+99] B. Demoen, M. García de la Banda, W. Harvey, K. Marriott, and P.J. Stuckey. An overview of HAL. In *Proc. of the Fourth International Conference on Principles and Practices of Constraint Programming*, LNCS, pages 174–188. Springer, October 1999.

[Duc] G. J. Duck. A CHR/WAM implementation for Haskell. http://www.cs.mu.oz.au/~ gjd/haskellchr/.

[Frü95] T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.

[GHC] Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

[GJDS04] 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.

[Has] Haskell 98 language report. http://research.microsoft.com/Users/simonpj/haskell98-revised/haskell98-report-html/.

[Hen01] F. Henderson et al. The Mercury language reference manual, 2001. http://www.cs.mu.oz.au/research/mercury/.

[HUG] Hugs home page. haskell.cs.yale.edu/hugs/.

[JJM97] S. Peyton Jones, M. P. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[Jon00] M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

[Läu96] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, 1996.

[Mah05] M. Maher. Herbrand constraint abduction. In *Proceedings of Logic in Computer Science Symposium (LICS)*, page to appear, 2005.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.

[Mil92] Dale Miller. Unification under a mixed prefix. *J. Symb. Comput.*, 14(4):321–358, 1992.

[Pei65] C.S. Peirce. *Collected Papers of Charles Saunders Peirce*. Belknap Press of Harvard University Press, 1965.

[SAC00] S S. Abdennadher and H. Christiansen. An experimental clp platform for integrity constraints and abduction. In *Proceedings of FQAS2000, Flexible Query Answering Systems*, Advances in Soft Computing series, Physica-Verlag, pages 141–152. Springer, 2000.

[SS02] P. J. Stuckey and M. Sulzmann. A theory of overloading. In *Proc. of ICFP'02*, pages 167–178. ACM Press, 2002.

[SW] M. Sulzmann and J. Wazny. Chameleon. http://www.comp.nus.edu.sg/~ sulzmann/chameleon.

[SWS05] M. Sulzmann, J. Wazny, and P. J. Stuckey. A framework for extended algebraic data types. http://www.comp.nus.edu.sg/~ sulzmann, 2005.

[Wan87] Mitchell Wand. A simple algorithm and proof for type inference. In *Fundamenta Informaticae X*, pages 115–122. North-Holland, 1987.

Logical Rules for a Lexicographic Order Constraint Solver

Thom Frühwirth

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

Abstract. We give an executable specification of the global constraint of lexicographic order in Constraint Handling Rules (CHR) language. In contrast to previous approaches, the implementation is short and concise without giving up on linear time worst case time complexity. It is incremental and concurrent by nature of CHR. It is provably correct and confluent. It is independent of the underlying constraint system, and therefore not restricted to finite domains. We also show completeness of constraint propagation, i.e. that all possible consequences of the constraint are generated by the implementation. Our algorithm is encoded by three pairs of rules, two corresponding to base cases, two performing the obvious traversal of the sequences to be compared and two covering a not so obvious special case when the lexicographic constraint has a unique solution.

1 Introduction

Lexicographic orderings are common in everyday life as e.g. the alphabetical order used in dictionaries and listings. Lexicographic orders also play a central role in termination analysis, for example for rewrite systems [3]. In constraint programming, these orders have recently raised interest because of their use in symmetry breaking (e.g. [15]) and earlier in modelling preferences among solutions (e.g. [8]).

Related Work. A natural question to ask is whether lexicographic orders can be implemented as constraints and what would be appropriate incremental propagation algorithms. There are two approaches to the problem, starting with [9] and [6]. Both consider the case of finite domain constraints and (hyper/generalized) arc consistency algorithms, while our work is independent of the underlying constraint system and achieves complete constraint progagation as well. All approaches, including ours, yield algorithms with a worst-case time complexity that is linear in the size of the lexicographic ordering constraint.

The algorithms and their derivation are quite different, however. In [9] an algorithm based on two pointers that move along the elements of the sequences to be lexicographically ordered is given. The algorithm is not incremental and its description consists of five procedures with 45 lines of pseudo-code. In [6], a case analysis of the lexicographic order constraints yields 7 cases to distinguish,

these are translated into a finite automaton that is then made incremental. The pseudo-code of the algorithm has 42 lines [5]. The manual derivation of the algorithm is made semi-automatic in a subsequent paper [4], that can deal with an impressive range of global constraints over sequences. The pseudo-code of a simple constraint checker is converted by hand into a corresponding automaton code (16 lines) that is automatically translated into automata constraints that allow incremental execution of the automaton and so enforce arc consistency. Note that the automaton code is interpreted at run-time.

We summarize that these approaches are based on imperative pseudo-code that is either lengthy or requires subsequent translation into a different formalism. These specifications are hard to analyse and are not directly executable. In contrast, we give a short and concise executable specification in the Constraint Handling Rules (CHR) language that consists of 6 rules that derive from three cases. The problem is solved by recursive decomposition, no auxiliary constraints are needed. The implementation is incremental and concurrent by nature of CHR. It is independent of the underlying constraint system, and therefore not restricted to finite domains. Its CHR rules can be analysed, for example we will show their confluence using a confluence checker, and prove their logical correctness. We show linear worst-case time complexity and that they propagate as much information (constraints) as is possible.

CHR [10, 14, 12] is a concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multi-sets of constraints (atomic formulae) into simpler ones until they are solved.

CHR was initially developped for writing constraint solvers, but has matured into a general-purpose concurrent constraint language over the last decade. Its main features are a kind of multi-set rewriting combined with propagation rules. The clean logical semantics of CHR facilitates non-trivial program analysis and transformation. Implementations of CHR now exist in many Prolog systems, also in Haskell and Java. Besides constraint solvers, applications of CHR range from type systems and time tabling to ray tracing and cancer diagnosis.

Overview of the Paper. After introducing CHR, we give our implementation in Section 3. Then, in separate sections, we discuss confluence, logical correctness, completeness and worst-case time complexity before we conclude. To the best of our knowledge, this is the first paper that discusses the implementation of a global constraint in CHR.

2 Preliminaries: Constraint Handling Rules

In this section we give an overview of syntax and semantics for constraint handling rules (CHR) [10, 14, 12]. Readers familiar with CHR can skip this section.

2.1 Syntax of CHR

We use two disjoint sets of predicate symbols for two different kinds of constraints: built-in (pre-defined) constraint symbols which are solved by a given

constraint solver, and CHR (user-defined) constraint symbols which are defined by the rules in a CHR program. There are three kinds of rules:

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

where *Name* is an optional, unique identifier of a rule, the l.h.s. *head* H, H' is a non-empty comma-separated conjunction of CHR constraints, the *guard* C is a conjunction of built-in constraints, and the r.h.s. *body* B is a goal. A *goal (query, problem)* is a conjunction of built-in and CHR constraints. A trivial guard expression "`true |`" can be omitted from a rule.

Simpagation rules abbreviate simplification rules of the form

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

2.2 Declarative Semantics of CHR

The CHR rules have an immediate logical reading, where the guard implies a logical equality or implication between l.h.s. and r.h.s. of a rule.

The logical meaning of a simplification rule is a logical equivalence provided the guard holds.

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

where $\bar{y}$ are the variables that appear only in the body B.

The logical meaning of a propagation rule is an implication provided the guard holds

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

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

2.3 Operational Semantics of CHR

The operational semantics of CHR is given by a transition system, where computation states are goals, i.e. conjunctions of built-in and CHR constraints.

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

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

When a simplification rule is applied, the matched constraints in the current goal are replaced by the body of the rule. When a propagation rule is applied,

the body of the rule is added to the goal without removing any constraints[1]. When a simpagation rule is applied, all constraints to the right of the backslash are replaced by the body of the rule.

3 A Lexicographic Order Constraint Solver

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

Definition 1. Given two sequences l_1 and l_2 of variables of the same length n, $[x_1, \ldots, x_n]$ and $[y_1, \ldots, y_n]$, then $l_1 \preceq_{lex} l_2$ iff $n{=}0$ or $x_1{<}y_1$ or $x_1{=}y_1$ and $[x_2, \ldots, x_n] \preceq_{lex} [y_2, \ldots, y_n]$.

This inductive definition[2] can be directly modeled as a recursive predicate in Prolog (where sequences are represented by lists). The built-ins `<` and `=` are either tests or constraints. In the latter case, also non-ground lists can be handled.

```
[] lex [].
[X|L1] lex [Y|L2] :- X<Y.
[X|L1] lex [Y|L2] :- X=Y, L1 lex L2.
```

This executable specification in Prolog relies on backtracking search to find the applicable clauses. A smart compiler may remove clause choices for ground lists, but if the relationship between some variable pairs is not known, search cannot be avoided.

The logical reading of this Prolog program, Clark's completion [7], almost literally corresponds to the above definition, hence the specification is easily seen to be *logically correct*.

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

3.1 A First Implemention in CHR

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

[1] To avoid non-termination, a propagation rule is never applied a second time to the same constraints.

[2] All what follows will also work with sequences of different length and with strict lexicographic order, but one has to add a few more rules. We have chosen the same lenghts in order to be able to concentrate on the essential rules for propagation.

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

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

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

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

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

and implement it as a CHR propagation rule, where the built-in inequality constraint appears in the body of a rule and is thus enforced when the rule is applied[3]

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

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

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

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

Admittedly, these last two rules require some insight.

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

Our example, `[R1,R2,R3] lex [T1,T2,T3], R2=T2, R3>T3`, can now be handled. First, since `R2=T2`, rule `l6` is applicable, and its result is `R2=T2, R3>T3`,

[3] It is the responsibility of the compiler to generate appropriate code for built-in constraints depending on their use in guards and bodies.

`[R1,R3] lex [T1,T3]`. Now rule `l5` is applied, and we arrive at `R2=T2, R3>T3, R1<T1` as desired.

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

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

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

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

Time Complexity. To show that worst-case time complexity is not linear[4], we consider the longest sequences of computation steps (rule applications) caused by the propagation rule `l6'`. Assume `[X,U1,...Un] lex [Y,V1,...Vn]` with `U1>=V1,...Un>=Vn`. Repeated recursive application of the propagation rule `l6'` will result in `n lex` constraints `[X,U1,...Un] lex [Y,V1,...Vn]`, `[X,U2,...Un] lex [Y,V2,...Vn]`,...`[X,Un] lex [Y,Vn]` with $O(\mathtt{n}^2)$ occurrences of variables.

If one of the variable pairs `Ui` and `Vi` is turned into a strict inequality `Ui>Vi`, the pair `X` and `Y` will be constrained to `X<Y`. This in turn will reduce all pending `lex` constraints (that all start with this variable pair) to `true`.

On the other hand, if `X=Y`, all propagated `lex`-constraints will be reduced by this pair of variables, `[U1,...Un] lex [V1,...Vn]`, `[U2,...Un] lex [V2,...Vn]`,...`[Un] lex [Vn]`. All subsequent pairs `Ui` and `Vi` will be forced to be equivalent due to the propagation rule `l4`, and the reduction will continue until all variable pairs are removed from all `lex` constraints. The result is `X=Y, U1=V1,...Un=Vn`.

In the last computation scenario, we have to remove $O(\mathtt{n}^2)$ occurrences of variables. The propagation rule `l6'` turns an otherwise linear complexity into a quadratic one. The problem is the redundant information (repeated variables) contained in each of the `lex`-constraints generated by the propagation rule.

3.2 Improving Time Complexity

In order to regain linear time complexity, we try to replace the propagation rule `l6'` by an equally powerful simplification rule. Just changing `==>` into `<=>`

[4] We assume a standard CHR implementation here. With specific optimizations, linear complexity is possible.

results in a rule that is logically incorrect and also results in non-confluence. A semantics-preserving translation of the propagation rule by adding the head to the body, `[X,U|L1] lex [Y,V|L2] <=> U>=V | [X,U|L1] lex [Y,V|L2], [X|L1] lex [Y|L2]`, leads to obvious non-termination. But it gives us a clue in that it shows the problem of the repated occurences of the subsequences `L1` and `L2`. As it turns out (see Section on Correctness), removing `L1` and `L2` from the added `lex` constraint preserves correctness and improves time complexity to linear.

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

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

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

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

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

As we will later see, the rules `l1` and `l2` are not needed for completeness of constraint propagation, but are useful for garbage collection, maintaining confluence and for detection of entailment (a constraint is entailed if it reduces to `true` without adding new constraints).

4 Confluence

Typically, CHR programs for constraint solving are well-behaved, i.e. terminating and confluent. Confluence means that the result of a computation is independent from the order in which rules are applied to the constraints. This also implies that the order of constraints in a goal does not matter. Once termination has been established [11], there is a decidable, sufficient and necessary test for confluence [1, 2]. In the latter papers it is also shown that confluent CHR programs have a consistent logical reading.

Definition 2. A CHR program is *confluent* if for all computation states S, S_1, S_2: If $S \mapsto^* S_1$ and $S \mapsto^* S_2$ then there exist states T_1 and T_2 such that $S_1 \mapsto^* T_1$ and $S_2 \mapsto^* T_2$ and T_1 and T_2 are identical up to renaming of local variables and logical equivalence of built-in constraints.

For checking confluence, one takes two rules (not necessarily different) from the program. The heads of the rules are overlapped by equating at least one

head constraint from one rule with one from the other rule. For each *overlap*, we consider the two states resulting from applying one or the other rule. These two states form a so-called *critical pair*. One tries to *join* the states in the critical pair by finding two computations starting from the states that reach a common state. If the critical pair is not joinable, we have found a counterexample for confluence of the program.

4.1 Confluence of the `lex`-Constraint

We used the confluence checker mentioned in [13]. The six rules for the lexicographic order constraint are confluent, the program code and its results are available at:
`www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/more/conflexico.pl`

The rule `l1` cannot give rise to any critical pair. It does not overlap with any other rule, since it is the only one dealing with empty lists. The rules `l2` and `l3` are mutually exclusive. There are overlaps between all the remaining pairs of rules.

For example, consider the overlap between the rules

```
l3 @ [X|L1] lex [Y|L2] <=> X=Y | L1 lex L2.
l5 @ [X,U|L1] lex [Y,V|L2] <=> U>V | X<Y.
```

which is

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

Using the first rule, we arrive at `X=Y, U>V, [U|L1] lex [V|L2]`. Using the second rule, we arrive at `X=Y, U>V, X<Y`. These two states form the critical pair. The propagation rule is applicable to the first state `X=Y, U>V, [U|L1] lex [V|L2]` and leads to `X=Y, U>V, [U|L1] lex [V|L2], U=<V`, which fails due to the contradicting constraints on `U` and `V`. The second state immediately fails due to the contradicting constraints on `X` and `Y`. Hence, this critical pair is joinable, in both cases we finally fail (independent of the order of rule applications).

Another interesting critical pair is between the rules `l5` and `l6''`.

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

Their overlap is

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

The resulting critical pair is

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

The first state of the critical pair is already a final state, in the second one, rule `l5` can be applied to the first `lex constraint` resulting in `U>V, L1=[_|_], X<Y, [X|L1] lex [Y|L2]`. Now, since `X<Y`, rule `l2` can be applied to remove the remaining `lex` constraint, so that the two states of the critical pair are the same. Hence it is joinable.

If rule `l2` or rule `l4` is dropped, the solver becomes non-confluent, while the other rules can be dropped without hurting confluence. Solver completeness is harmed if rules other than `l1` and `l2` are dropped. (This means that we could drop rule `l1` from the solver.)

5 Logical Correctness

The logical reading of the six rules for the lexicographic order constraint is as follows.

$$\begin{array}{rll} & ([]\preceq_{lex}[]) & \\ X{<}Y \rightarrow & ([X|L1]\preceq_{lex}[Y|L2]) & \\ X{=}Y \rightarrow & ([X|L1]\preceq_{lex}[Y|L2] & \leftrightarrow L1\preceq_{lex}L2) \\ & ([X|L1]\preceq_{lex}[Y|L2] & \rightarrow X{\leq}Y) \\ U{>}V \rightarrow & ([X,U|L1]\preceq_{lex}[Y,V|L2] & \leftrightarrow X<Y) \\ (U{\geq}V \wedge L1{=}[a|b]) \rightarrow & ([X,U|L1]\preceq_{lex}[Y,V|L2] & \leftrightarrow \\ & ([X,U]\preceq_{lex}[Y,V] \wedge [X|L1] \preceq_{lex}[Y|L2])) & \end{array}$$

It is easy to show that these formulas are logical consequences of the logical specification given by

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

As an example, we prove that the logical reading of rule `l6''` is a logical consequence of the logical specification of the `lex` constraint.

Logical Correctness Proof for Rule `l6''`. From the specification it follows

$$[X|L1]\preceq_{lex}[Y|L2] \leftrightarrow (X{<}Y \vee X{=}Y, L1\preceq_{lex}L2)$$

For the rule `l6''`, we will actually prove a slightly stronger result by removing $L1{=}[a|b]$ from the precondition (the condition was introduced to ensure termination). Instead of $C \rightarrow (H \leftrightarrow B)$ we use the equivalent $(H \wedge C) \leftrightarrow (C \wedge B)$). We will replace the `lex` constraints in the logical reading of the rule according to the specification, distribute conjunction over disjunction and simplify by removing unsatisfiable disjuncts.

According to the specification, the l.h.s. of the rule, $[X,U|L1]\preceq_{lex}[Y,V|L2] \wedge U{\geq}V$, becomes

$$\begin{array}{l} U{\geq}V \wedge (X{<}Y \ \vee \ X{=}Y \wedge U{<}V \ \vee \ X{=}Y \wedge U{=}V \wedge L1\preceq_{lex}L2) \leftrightarrow \\ (U{\geq}V \wedge X{<}Y \ \vee \ X{=}Y \wedge U{=}V \wedge L1\preceq_{lex}L2) \end{array}$$

The r.h.s. $U{\geq}V \wedge [X,U]{\preceq_{lex}}[Y,V] \wedge [X|L1]{\preceq_{lex}}[Y|L2]$ becomes

$U{\geq}V \wedge (X{<}Y \vee X{=}Y \wedge U{<}V \vee X{=}Y \wedge U{=}V) \wedge (X{<}Y \vee X{=}Y \wedge L1{\preceq_{lex}}L2) \leftrightarrow$
$U{\geq}V \wedge (U{\geq}V \wedge X{<}Y \vee X{=}Y \wedge U{=}V) \wedge (U{\geq}V \wedge X{<}Y \vee U{\geq}V \wedge X{=}Y \wedge L1{\preceq_{lex}}L2)$
$\leftrightarrow (U{\geq}V \wedge X{<}Y \ \vee \ X{=}Y \wedge U{=}V \wedge L1{\preceq_{lex}}L2)$

Both sides, l.h.s. and r.h.s., are equivalent.

6 Worst-Case Time Complexity

We would like to give a complexity result that is independent from the constraint system in which the built-in constraints (inequalities) are defined. The reason is that most constraint systems, such as finite domains, linear polynomials, Booleans, admit these inequalities, and that the typical algorithms used (e.g. arc and path consistency, simplex) have different time complexities.

We therefore give our complexity result in the number of atomic built-in constraints that are checked and imposed, respectively.

For the rules of the `lex` constraint, head matching can be done in constant time, guards contain at most one built-in inequality constraint to check, and rule bodies directly impose at most one built-in inequality constraint. Hence the number of checks and additions of built-in constraints is proportional to the number of rule applications.

To empty lists, only the rule `l1` is applicable. To lists with one or two elements, the rules `l2`, `l3`, `l4` and `l5` are applicable (but not rule `l6''`). The propagation rule `l4` can be applied exactly once to each `lex` constraint. Hence for lists of length less than or equal to 2, the number of rule applications is bounded by a constant (at most 5). Therefore the first recursive call of rule `l6''` involving a list of length 2 can only contribute a constant number of rule applications. Thus, for lists of length greater than 2, the number of rule applications is linear in the length of the list. Therefore the number of checks and additions of built-in inequality constraints is also linear in the length of the list.

In most constraint systems, the canonical form for constraints will require that all variables in a global constraint are different from each other (equivalent variables are equated by additional equality constraints). In that case, we can replace list length in the complexity result by the number of different variables in the list.

7 Completeness

In this section we discuss completeness of the constraint solver for the lexicographic order constraint, i.e. if it produces all built-in constraints, i.e. inequalities, that can be derived from the `lex` constraint.

A *solution* of a lexicographic order constraint $[x_1,\ldots,x_n]{\preceq_{lex}}[y_1,\ldots,y_n]$ is a conjunction of built-in inequality constraints that logically implies the order constraint. In practice, a solution can be obtained by unfolding the `lex` constraint

away, as is done in the Prolog implementation of the constraint. It is easy to see that a solution will be of the form

$$x_1{=}y_1 \wedge x_2{=}y_2 \wedge \ldots \wedge x_{i-1}{=}y_{i-1} \wedge x_i{<}y_i \quad (1 \leq i \leq n{+}1),$$

where $x_i{<}y_i$ is dropped from the conjunction if $i = n{+}1$. Hence there can be at most $n + 1$ solutions to a given `lex` constraint over lists of length n.

We describe such a solution by an expression $(=)^{i-1}[<]$. The resulting sequence of inequalities is meant to hold between the respective pairs of variables from the two lists of the `lex` constraint. $[e]$ means that expression e is dropped if its position in the sequence is greater than n. With $x?y$ we will denote $x < y \vee x = y \vee x > y$, which always holds for total orders that we use. The symbol ; denotes a choice between alternatives.

Next we convince ourselves that there are `lex` constraints over lists of length n that have as solutions any subset of the $n + 1$ possible solutions. For example, given three positions $1 \leq i < j < k \leq n{+}1$, any `lex` constraints whose list elements satisfy the inequality constraints specified by the expression $(\geq)^i?(\geq)^{j-i}?(\geq)^{k-j}[(< \ ; \ ?(\geq)^* >)]$ will exactly have the three desired solutions $(=)^i <$, $(=)^j <$, and $(=)^k[<]$.

The expression allows to construct a `lex` constraint with the desired solutions. The idea is to leave those positions unconstrained where a solution should end (denoted by ? in the expression), and to constrain all other positions to $\geq$, so that a solution ending at that position is not possible, since it would require $<$. But equality $=$ is possible, which allows to pass over that position and enable solutions that go beyond that position. In addition, special care has to be taken with the last solution. In order to avoid a solution that equates all variable pairs, the last position for a solution either has to be constrained to $<$, so that it cannot be passed over, or a constraint on a later position has to be imposed so that passing over cannot lead to a solution, this is the case if at some later position we impose $>$ instead of $\geq$. This is described by the alternative $?(\geq)^* >$ at the end of the expression. Note that it implies that ? must be constrained to $<$ if we want to have a solution at that position.

Clearly, the disjunction of all solutions of a `lex` constraint is logically equivalent to the constraint. The new inequalities that we can propagate from such a disjunction, i.e. those built-in constraints that are implied, that must hold no matter which disjunct (solution) is chosen, are simply and only $(=)^i \leq$, where i is the position of the first solution. A special case arises if there is only one solution, in that case obviously $(=)^i[<]$. As discussed above, this case arises if the initial expression is $(\geq)^*[?(\geq)^* >]$.

These observations mean that a complete implementation has to turn leading $\geq$ inequalities into equalities $=$ and impose $\leq$ on the first other inequality. In our constraint solver implementation this is achieved by the propagation rule `l4` that imposes $\leq$ on any current first position and the recursive simplification rule `l3` that removes leading $=$. The special case is handled by the simplification rules `l5` and `l6''`. Rule `l5` covers the case where $>$ holds for the second position, so $<$ must hold for the first position to ensure a solution. Rule `l6''` allows to reduce

the other instances of the special case, where there is an arbitrary number of $\geq$ constraints between the unique position for a solution and the $>$ inequality, to the situation in rule `l5`. (The rules `l1` and `l2` are not needed for completeness, but are useful simplifications to detect entailment.)

8 Conclusions

Just six CHR rules correctly and efficiently specify and implement an incremental, concurrent, logical algorithm to maintain consistency of the lexicographic ordering constraint. Previous approaches presented algorithms for the lexicographic order constraint in often lengthy pseudo-code that is hard to analyse, while our 6 rules are simple and directly executable. Moreover, our implementation is independent of the underlying constraint system, and therefore not restricted to finite domains. We have proven the rules to be confluent using our automatic confluence checker. We showed logical correctness, completeness of constraint propagation and linear worst-case time complexity.

Future work should give an implementation that does not rely on built-in constraints for inequality, but rather uses existing CHR solvers (e.g. for finite domains). Experimental evaluation would be next to find out the constant factors hidden in the time complexity. Extensions of the lexicographic ordering constraint that can be found in the recent literature, e.g. using it in chains or with a summation constraint, or simplifying `lex` constraints for symmetry breaking, could be considered. Finally, a hard, challenging question is if and how rules such as the ones presented here can be derived automatically from inductive definitions.

Acknowledgements. The author would like to thank Marc Meister, Tom Schrijvers and the reviewers for their comments.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *3rd International Conference on Principles and Practice of Constraint Programming*, LNCS 1330, Berlin, Heidelberg, New York, 1997. Springer.
2. S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal, Special Issue on the 2nd International Conference on Principles and Practice of Constraint Programming*, 4(2):133–165, 1999.
3. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge Univ. Press, 1998.
4. N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In M. Wallace, editor, *CP'2004, Principles and Practice of Constraint Programming*, volume 3258 of *LNCS*, Berlin, Heidelberg, New York, 2004. Springer.
5. M. Carlsson and N. Beldiceanu. Revisiting the lexicographic ordering constraint. Technical Report T2002-17, Swedish Institute of Computer Science, 2002.

6. M. Carlsson and N. Beldiceanu. From constraints to finite automata to filtering algorithms. In D. Schmidt, editor, *ESOP2004*, volume 2986 of *LNCS*, pages 94–108, Berlin, Heidelberg, New York, 2004. Springer.
7. K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.
8. H. Fargier, J. Lang, and T. Schiex. Selecting preferred solutions in fuzzy constraint satisfaction problems. In *1st European Congress on Fuzzy and Intelligent Technologies (EUFIT)*, 1993.
9. A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In P. V. Hentenryck, editor, *CP'2002, Int. Conf. on Principles and Practice of Constraint Programming*, volume 2470 of *LNCS*, pages 93–108, Berlin, Heidelberg, New York, 2002. Springer.
10. 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.
11. T. Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In *8th International Conference on Principles of Knowledge Representation and Reasoning*, Toulouse, France, 2002.
12. T. Frühwirth. CHR web-pages, www.informatik.uni-ulm.de/pm/ mitarbeiter/fruehwirth/chr.html, 2004.
13. T. Frühwirth. Parallelizing union-find in constraint handling rules using confluence. In *International Conference on Logic Programming (ICLP)*, LNCS, Barcelona, Spain, 2005. Springer.
14. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.
15. B. Hnich, Z. Kiziltan, and T. Walsh. Combining symmetry breaking with other constraints: Lexicographic ordering with sums. In *AMAI 2004 Eighth International Symposium on Artificial Intelligence And Mathematics*, 2004.

Reasoning about passive declarations in CHR

Henning Christiansen

Roskilde University, Computer Science Dept. DK-4000 Roskilde, Denmark
E-mail: henning@ruc.dk

Abstract. The programming language of Constraint Handling Rules (CHR) is gaining more and more popularity and this has motivated the development of new optimization techniques to be applied in implementations of CHR. As for other logic programming languages, a program written CHR can be understood declaratively as a logical formula and as a procedural specification, and CHR has different tools for manual optimization at the procedural level whose application may (or may not) sacrifice the declarative reading. One such optimization is given by passive declarations by means of which the search for rules to be applied can be optimized, perhaps changing which rules that are applied and in which order. A framework for analyzing the effect of passive declarations is presented in terms of a slightly abstract operational semantics which takes into account the effect of passive declarations. Based on it, we can give different classifications of the effect of some passive declarations that do not affect the logical semantics. This may be used for reasoning about manual optimizations as well as proposals for automatic strategies to be applied by a compiler for adding passive declarations.

1 Introduction

The programming language of Constraint Handling Rules [4] (CHR) was originally introduced as a specialized tool for writing constraint solvers for standard domains such as real and rational arithmetics, finite domains, etc., but has gained a growing popularity as a programming language of a wider scope. Implementations of CHR are now available in a number of Prolog systems (e.g., [14]) and other platforms,[1] and the development of program analysis and optimization methods for CHR is currently an active research area.

CHR includes passive declarations by means of which a programmer can affect the search for rules to be applied in order to obtain a faster program execution. Passive declarations may or may not affect the declarative semantics of a program, and we present an operational semantics of CHR that captures the effect of passive declarations, and by means of which we can characterize some classes of programs where the declarative semantics is preserved. This framework can be used for reasoning about manual optimizations as well as strategies to

[1] The web page http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/ refers at the time of writing to nine implementations embedded in Prolog, one in Haskell and one in Java.

be applied in CHR compilers or high-level source-to-source translators; as an example of the latter, we consider optimizations applied in the CHRG system [2] which translates a grammar notation into CHR.

Recently, other authors have proposed what they call a refined operation semantics [3] which is intended to capture all procedural aspects of actual CHR implementations, and as such it is completely deterministic. By nature, this refined semantics is very complicated and detailed, and for our own purposes, which includes doing proofs by hand, we have chosen a more abstract version which includes sufficient details to characterize essential properties of passive declarations, but is nondeterministic for the rest; we have, however, not made a formal comparison of the two semantics.

As will be shown below, passive declarations of a program in general can only preserve the semantics for a specific class of queries. The current CHR syntax does not recognize such restrictions to classes of queries and in this way, our methodology may in some cases go further than what is possible in existing compilers.

Over the last 50 years, the need for efficient compilers has driven the development of sophisticated methods for control and data flow analyses, including abstract interpretation, for other programming languages, including traditional logic programming languages such as Prolog. For CHR, on the other hand, such analyses tend to be quite difficult due to the demon-like nature of rules, and it is only very recently that such methods have been suggested; references are given below.

Based on our operational semantics, we define an important class of *control-safe* passive declarations that do not change the order in which rules are applied, but inform the CHR compiler where optimization can be made. We indicate also a wider class of *answer-safe* passive declarations which may change flow of control but do not change result of the computation.

In section 2 we review informally the operational semantics of CHR and discuss the meaning of passive declations, and the two notions of safety are defined; section 3 presents the operational semantics. In section 4, we present a method for proving control-safety based on local properties of each rule; at present it is not suited for automatic systems as it requires the invention of a state invariant. Classes of programs, for which we can point to patterns of passive declarations guaranteed to be control-safe, are given in section 5, CHRs for parsing and a class of programs with a separation of constraints into data structure and process constraints. Section 6 considers the answer-safe (and not necessarily control-safe) passive declarations for programs that have been proved to be confluent. Here a simple criterion based on subsumption of rule heads applies. Section 7 analyzes the speedup gained by passive declarations and provides a few benchmark tests. Sections 8 and 9 review briefly related works and give conclusion and perspectives.

2 CHR and passive declaration

For an introduction to CHR and its declarative semantics, we refer to [4]. Similarly to [3] we consider only simpagation rules, as they subsume both propagation and simplification rules. The general form is as follows; P, Q, and R are sequences of constraints, G a guard which is a test on variables of P and Q made by built-in test predicates. For simplicity, we allow only the built-in constraint "=" with the usual meaning of unification in R (but not in P and Q).[2]

$$P \setminus Q \iff G \mid R$$

The left hand side $P \setminus Q$ is called the *head* of the rule and R the *body*. In case P is empty, the backslash is left out and the rule is called a simplification rule. When Q is empty, we talk about a propagation rule, leaving out again the backslash, but replacing the arrow symbol with ==>. Procedurally, a CHR program can be understood basically as the rewriting of constraint stores with the initial constraint store given by a query. A simpagation rule as above can be applied whenever constraints matching P, Q are found in the store and G is satisfied; in that case, constraints matched by Q are removed from the store and the corresponding instance of R is added. This principle can with a few simple adjustments be formalized into an operational semantics which satisfies standard correctness properties wrt. a declarative semantics. Unfortunately, the statements provided by the declarative semantics or the operational one indicated above, are often very weak, and it may be claimed that CHR programmers tend to work with the *de-facto* operational semantics given by the current implementations (e.g., the version in SICStus Prolog [14] considered the reference implementation) in mind rather that the logical one. As noted by [3], even example programs in the reference manual of CHR can only be explained by referring to the *de-facto* operational semantics. This motivated [3] to describe a so-called refined operational semantics that gives the precise control flow of the *de-facto* operational semantics. The following properties are essential in the refined semantics:

1. Calls of constraints are procedure calls. When a constraint c is called and it triggers some rule, then the constraints in that rule's body are called recursively, triggering other rules, and control can only return to c when these other rules have finished (unless c has been consumed by a simplification/simpagation step).
2. Left-to-right execution. The constraints in a query or rule body are called in a sequential manner from left to right.
3. When a constraint c is called, rules containing the same predicate as c are tested in their textual order for possible application.

[2] In versions of CHR embedded in Prolog. e.g., [14], the guard may also refer to user-defined predicates; we ignore the possible splitting of the guard into an ask and tell part. Typically, the body R can be any callable Prolog term which will make analysis more complicated but does not change the overall mechanisms.

4. When a constraint c is called, it gets the status of "operational" meaning that other rules triggered directly or indirectly by c can access c and perhaps even delete it by means of a simplification rule. (Intuitively, an operational constraints such as c belongs at this point to both the constraint store and to the recursion stack.)

Example 1. The last property can be illustrated by the following CHR program which has been tested in a running implementation of CHR.

```
a    ==>  write('rule1 '), b.
a, b <=>  write('rule2 '), c.
b    <=>  write('rule3 ').
```

A call to `a` results in the printout `rule1 rule2` and a final constraint store consisting of `c`. The call to `a` gives rise to a call to `b`; the first rule with `b` in the head is tried, and since `a` is accessible as "operational", this rule can fire, thus consuming `a` and `b` as it is a simplification rule.

Constraints in the head of a rule may be declared passive using the following syntax.

```
a(X)#Id, b(X) ==> c(X) pragma passive(Id)
```

The rule behaves like `a(X), b(X) ==> c(X)` except that, when a call `a(···)` tries all rules in textual order, this one is skipped. Thus this rule can only be triggered by a call of form `b(···)` and, of course, when a suitable operational `a(···)` can be located. It will be a straightforward matter to incorporate this behaviour in the refined semantics [3] which we shall refrain from here.

¿From now on we abbreviate the syntax for passive declarations to just the `#` symbol on the head constraints. To simplify notation, we indicate a set of passive declaration of a specific program as an operator $\mathcal{P}$, and when writing $\mathcal{P}(\Pi)$ for program Π, it is assumed that Π refers to the version without passive declarations; similarly, for a rule $\mathcal{P}(\rho)$ of $\mathcal{P}(\Pi)$, the argument ρ refers to the matching rule of Π. It is understood that rules and elements of rules occur in a specific order and that the elements of $\mathcal{P}$ refer to specific positions in rule heads.

Assume, now, that the sample rule above is part of a program which has the overall property that whenever a call `a(`t`)` is made operational during execution, there cannot be any matching `b(`t`)` operational in the same state. Then, clearly, the passive declaration does not affect the order in which the rules are applied and thus the final result of the computation. This is an instance of what we define below as control-safety. In general, however, such a properties can only be ensured under given restrictions to the input queries. If, for example, the query `b(1),a(1)` is possible, the desired property does not hold.

A semiformal definition is sufficient for our purposes, based on the understanding of an operational semantics being a device that takes a query (a sequence of constraints) plus program as input and generates a set of derivations being sequences of states with each transition indicated by the instance of the specific rule being applied. A derivation has an initial state isomorphic to a query, and may be infinite or have a final state from which a computed answer is abstracted.

Definition 1. *Assume an operational semantics for CHR, a program $\mathcal{P}(\Pi)$ and a set of queries Θ. Then the passive declarations $\mathcal{P}$ of $\mathcal{P}(\Pi)$ are said to be*

- control-safe *for Θ whenever the set of derivations for queries in Θ is independent of whether program Π or $\mathcal{P}(\Pi)$ is used, and*
- answer-safe *for Θ whenever the set of pairs*
  $\{\langle Q, \textit{computed answer for } Q\rangle \mid Q \in \Theta\}$
  is independent of whether program Π or $\mathcal{P}(\Pi)$ is used.

Control-safety implies answer-safety but not vice versa. For simplicity, we do not consider the case where the user has supplied some passive declarations already before analysis takes place. It is, however, straightforward to generalize our definitions and constructions to handle these cases, except for those of section 6 for confluent programs.

3 An operational semantics

The simpler semantics, the easier analysis, and here we propose a semantics which resembles the refined semantics but reintroduces nondetermiminism in certain steps, while retaining the essential properties concerning passive declarations.

We abstract away here, the textual order of rules, the order in which the state is searched for operational constraints when a call is looking for companions to form a head match, and the bookkeeping machinery that is needed to avoid loops by propagation rules applying to the same constraints over and over.

Any constraint in the state is classified as either *operational* or *unseen*, indicated by means of a respective subscript $^\bullet$ or $^\circ$. A state is a pair of a recursion stack being a finite sequence of constraints (operational and unseen mixed) and a constraint store which is a multiset of operational constraints.

We assume a fixed program given, and by an *application instance* of a rule $P \setminus Q \mathrel{<=>} G \mid R$ we mean an instance extended with operational and unseen markings as follows

$$(P^\bullet \setminus Q^\bullet \mathrel{<=>} G \mid R^\circ)\sigma$$

where σ is any substitution with the property that it assigns a unique and unused variable to each variable of G and R which is not in P or Q, and so that $\models \exists\bar{x}(G\sigma)$ where $\bar{x}$ are the variables of $G\sigma$ not in $(P,Q)\sigma$. This definition is intended to simplify the semantics below in that head constraints of application instances may be identical to other operational constraints in the state (and thus CHR's notion of matching is handled in this way). Body constraints of application instances are marked as unseen with the intension that they stay as such until changed by an explicit step in the semantics. Notice that individual head constraints may be marked as passive as well; to simplify notation, passive markings are immaterial when constraints are compared by equality or subset operations. The symbols $\uplus$ refers to multiset union (with sequences viewed as multisets when relevant) and $\sqsubseteq$ to multiset inclusion.

A given query q is executed by a call $\mathsf{Run}(q^\bullet, \emptyset)$. This procedure, which is nondeterministic, is described as follows; ϵ denotes the empty stack, a dot is used as combined construction and concatenation operator. Letter A refers to a single constraint, and other letters to sequences or multisets as indicated by context.

1. **(halt)** $\mathsf{Run}(\epsilon, C^\bullet)$ halts for any $C^\bullet$.
2. **(simplify)** $\mathsf{Run}(A^\bullet \cdot q, C^\circ) \equiv$
   Select an application instance of a rule
   $P^\bullet \backslash Q_1^\bullet, A^\bullet, Q_2^\bullet \mathsf{<=>} G | R^\circ$
   with $P^\bullet \uplus Q_1^\bullet \uplus Q_2^\bullet \trianglelefteq C^\bullet \uplus q$ and no passive mark on $A^\bullet$, and continue with
   $\mathsf{Run}(R^\circ \cdot q', C'^\bullet)$
   where q' and $C'^\bullet$ appear by removing each element of $Q_1^\bullet \uplus Q_2^\bullet$ exactly once from either q or $C^\bullet$.
3. **(propagate)** $\mathsf{Run}(A^\bullet \cdot q, C^\bullet) \equiv$
   Select an application instance of a rule
   $P_1^\bullet, A^\bullet, P_2^\bullet \backslash Q^\bullet \mathsf{<=>} G | R^\circ$
   with $P_1^\bullet \uplus P_2^\bullet \uplus Q^\bullet \trianglelefteq C^\bullet \uplus q$ and no passive mark on $A^\bullet$, and continue with
   $\mathsf{Run}(R^\circ \cdot A^\bullet \cdot q', C'^\bullet)$
   where q' and $C'^\bullet$ appear by removing each element of $Q^\bullet$ exactly once from either q or $C^\bullet$.
4. **(activate)** $\mathsf{Run}(A^\circ \cdot q, C^\bullet) \equiv \mathsf{Run}(A^\bullet \cdot q, C^\bullet)$
5. **(shift)** $\mathsf{Run}(A^\bullet \cdot q, C^\bullet) \equiv \mathsf{Run}(q, C^\bullet \uplus \{A^\bullet\})$
6. **(solve)** $\mathsf{Run}((s = t)^\bullet \cdot q, C^\bullet) \equiv \mathsf{Run}((seq(R^\circ) \cdot q')\sigma, C'^\bullet\sigma)$
   where σ is an mgu of s, t and $R^\bullet$ the largest multiset of constraints affected[3] by σ so that $R^\bullet \trianglelefteq q \uplus C^\bullet$, and q' and $C'^\bullet$ appear by removing each element of $R^\bullet$ exactly once from either q or $C^\bullet$; $seq(R^\circ)$ is an arbitrary ordering of the elements of R° into a sequence.

Compared with [3], the choice of possible rule to apply is nondeterministic as well as when the shift step applies, e.g., for stopping a loop by the same propagation rule applying over and over again. The solve step is nondeterministic in the order in which the affected constraints are re-entered onto the recursion stack. Notice that the invention of application instances for steps 2 and 3 ensures that the complicated solve step only is involved in case the rule bodies or the initial query contains unifications.

4 Control-safety checked by local properties

Control-flow analysis and abstract interpretation for CHR programs are still a relatively sparsely investigated field, and another way to check control-safety is to check a slightly stronger property that can be checked locally for each step made in the operational semantics. However, this method needs the invention

[3] A substitution σ affects a variable x whenever $x\sigma \neq x$, and it affects any constraint containing a variable affected by σ.

of an invariant property for states, and it should also be proved that each step preserves the invariant.

Definition 2. *Let $\mathcal{P}(\Pi)$ be a program with passive declarations and* Inv *some property which may or may not hold for an execution state. A rule $\mathcal{P}(\rho) \in \mathcal{P}(\Pi)$ is* control-safe wrt *Inv whenever:*

- *For any state Σ satisfying* Inv*, $\mathcal{P}(\rho)$ can apply in Σ (step 2 or 3 of* Run*) if and only if ρ can apply, and such cases, the resulting successor state satisfies* Inv*.*

In case any rule $\mathcal{P}(\rho) \in \mathcal{P}(\Pi)$ is control-safe wrt Inv *and steps 4–6 (activation, shift, and solve) preserve* Inv*, we say that $\mathcal{P}(\Pi)$ is* locally control-safe *with respect to* Inv*.*

Control-safety for each rule can be checked by looking at that rule in isolation just assuming Inv and similarly for the other conditions of definition 2.

The following central property follows by a straightforward induction proof.

Proposition 1. *Let $\mathcal{P}(\Pi)$ be a program with passive declarations that are locally control-safe wrt* Inv*. Then the passive declarations of $\mathcal{P}(\Pi)$ are control-safe for the set of queries q for which $\langle q^\circ, \emptyset \rangle$ satisfies* Inv*.*

In other words, if we can design an invariant which accepts the desired class of queries and for which local safety can be shown for a progran $\mathcal{P}(\Pi)$, we have a guarantee that the indicated passive declarations do not change the semantics of Π, so we can let the CHR compiler optimize rule search based on those passive declarations.

5 Classes of programs with general strategies for control-safe passive declarations

5.1 CHR programs for bottom-up parsing

The CHR Grammar system (CHRG) [2] adds a grammar notation on top of CHR analogous to the way Definite Clause Grammars are added on top of Prolog. A CHRG is automatically compiled into a CHR program that works as a bottom-up parser and the system optionally adds passive declarations according to a predefined pattern. We take this as an example to illustrate how the method outlined in section 4 can be used to show the correctness of such a general strategy for a class of programs.

Example 2. The following CHR program implements a bottom-up parser for a small language; notice the passive declarations on all but rightmost head constraints.

```
np(N0,N1)#, v(N1,N2)#, np(N2,N3) ==> s(N0,N3).
token(N0,N1,peter) ==> np(N0,N1).
token(N0,N1,mary)  ==> np(N0,N1).
token(N0,N1,likes) ==>  v(N0,N1).
```

The parsing of a string can done by an initial query such as the following:

```
token(0,1,peter), token(1,2,likes), token(2,3,mary)
```

Let LRQ be the set of all queries of the form indicated by 2, consisting of `token` constraints, where the two first arguments recur as indicated and where the third argument can be whatever, so that it can be seen as a text string entered in left to right order.

We generalize the LRQ property into a state invariant by saying that a state $\langle q, C^\bullet \rangle$ satisfies Inv_{LR} whenever there exists a natural number $\mathbf{n}$ so that

- any constraint in the state is of form $c(n, m, \ldots)$ where n, m are natural numbers with $n < m$,
- any constraint in $C^\bullet$ and any operational constraint in q is of the form $c(n_0, n_1, \ldots)$ with $n_1 \leq \mathbf{n}$,
- Q is a sequence of the form $S \cdot U$ where S consists of operational constraints and U of unseen constraints; any constraint in S is of form $c(k, \mathbf{n}, \ldots)$; U is of the form $\langle \mathtt{token}(\mathbf{n}+1, \mathbf{n}+2, \ldots), \mathtt{token}(\mathbf{n}+2, \mathbf{n}+3, \ldots), \ldots \rangle$

With this, we can prove local control-safety for the program of example 2 from which follows control-safety.

In general, the CHRG system allows counterparts to propagation, simplification, and simpagation rules of CHR and allows also context-sensitive rules that may refer to symbols to the left and right of the string matched.

The proof of local control-safety for the sample program generalizes immediately to all CHR programs that results from the compilation of programs without right context, and with passive declarations applied to all but right-most constraints in each rule. See the detailed arguments in [2] which, however, does not use the framework of the present paper. In section 7 we report experiments measuring the actual speedup gained by passive declarations for such programs.

5.2 Programs with data structure and process constraints

Consider a class of CHR programs for which constraints can be separated into two disjoint classes D that we call *data structure constraints* and P called *process constraints*; let d and p be regular expression describing the respective classes of D and P constraints. Queries are restricted to the form d^*p^*. Each rule is of the form (commas implicit; notice that the order of constraints in heads is insignificant; for simplicity we show only propagation rules but the idea generalizes easily):

$$d^*p^+ \texttt{==>} p^*$$

When any D constraint is declared passive, the result is obviously control-safe.

In this case, the speedup is limited as it only affects the entering of the d^* part of the query into the constraint store. An example of the indicated pattern may be that d^* defines the edges of a graph and p^* are intended to check properties of that graph. We may generalize the pattern by introducing an intermediate class called F (decribed by regular expression f) of *formatting constraints* which, in

the graph example, could evaluate the transitive closure of d^* corresponding to the paths in the graph. Rules are now of the forms

$$d^* f^+ \texttt{==>} f^* \qquad \text{and} \qquad d^* f^* p^+ \texttt{==>} p^*$$

In the first kind, d constraints can be made passive and in the second both d and f, and the result is also control-safe. Here the effective speedup should be expected to be significant (although our empirical tests below show something different). The program will execute in three phases, first D constraints are entered into the store with no rules checked, next the F constraints are derived but with no useless checks of applicability of rules referring to P; finally the P constraints execute with same efficiency independent of the passive declarations.

6 Confluent programs

Confluent CHR programs have the property that the final state is independent of the order in which rules are applied. Precise definitions and ways of showing confluence of programs are given by [1]. In other words, any possible derivation for a confluent program is, so to speak, equally good, and possible derivations can be disregarded as long as one derivation remains. ¿From this argument, it follows that control-safety is an unnecessarily strong restriction for achieving answer-safety for confluent programs.

There is, in fact, a straightforward criterion that ensures answer-safety for programs proved by other means to be confluent.

Definition 3. *Let $\mathcal{P}$ be a set of passive declarations of a program $\mathcal{P}(\Pi)$ where Π is confluent. We say that $\mathcal{P}$ is* non-blocking *if, in any state in which a rule of Π can apply, there is some rule of $\mathcal{P}(\Pi)$ that can apply.*

This condition is obviously sufficient for answer-safety, and it is also necessary. To see this, assume that the passive declarations of a program $\mathcal{P}(\Pi)$ are not non-blocking, i.e., there is a combination of constraints Q for which no rule of $\mathcal{P}(\Pi)$ can apply, but there is one in Π that can apply. Obviously, with Q as query, the final state reached by $\mathcal{P}(\Pi)$ is different from the one reached by Π.[4]

It should be kept in mind, however, that non-blocking passive declarations added to a confluent program may not be an optimization, but may in worst cases slow down execution considerably. This will be the case whenever the actual implementation of a confluent program Π chooses a short derivation for some query but $\mathcal{P}(\Pi)$ bypasses this option and goes a long and laborious way to find the same result. The opposite may, of course, also happen. There is a simple criterion for testing non-blocking, one rule at a time.

Proposition 2. *Let $\mathcal{P}$ be a set of passive declarations of a program $\mathcal{P}(\Pi)$ where Π is confluent. Then $\mathcal{P}$ is non-blocking whenever for any rule $\mathcal{P}(\rho) \in \mathcal{P}(\Pi)$ and*

[4] In this argument, we have assumed the absence of rules that do not affect the state such as `a,b==>true`.

any passive constraint c_ρ in its head, there is some rule $\mathcal{P}(\pi) \in \mathcal{P}(\Pi)$ for which the following hold; h_ρ and h_π refer to the heads of $\mathcal{P}(\rho)$ and $\mathcal{P}(\pi)$, and G_ρ and G_π to their guards:

- *The predicate of c_ρ occurs non-passively in the head of π as c_π.*
- *It holds that $\exists(c_\rho \wedge G_\rho) \rightarrow \exists(c_\pi \wedge G_\pi)$ and $\exists(h_\rho \wedge G_\rho) \rightarrow \exists(h_\pi \wedge G_\pi)$.*

The separate test for c_ρ, c_π in the last condition is only necessary in case π has another and passive occurrence of the predicate of c_ρ in its head. Notice that all tests can be made by a syntactic subsumption check combined with a constraint solver for the built-in predicates applied in the guards.

Example 3. Assume the following two rules are part of a confluent program that has been extended with passive declarations.

```
a(X), b(7)# ==> p(X).
b(X) <=> X =< 10 | q(X).
```

The passive declaration is non-blocking in the sense of the proposition since we have $\exists(\texttt{a(X)} \wedge \texttt{b(7)}) \rightarrow \exists(\texttt{b(X)} \wedge \texttt{X=<10})$. (Proper definitions of predicates `p` and `q` are expected for this program to be confluent.)

Example 4. In a rule with symmetry in the head such as

```
p(X), p(Y) ==> p(X,Y).
```

any of the two head constraints can be made passive (but not both), and we will have the non-blocking condition satisfied with π being the same rule as ρ. In fact, such an optimization is made already in existing CHR compilers, cf. [7].

The opposite of proposition 2 is not the case in general.

Example 5. Consider the following program.

```
p(1) <=> q(1).   p(X) <=> X\=1 | q(X).   p(X)# <=> q(X).
```

Its passive declaration is non-blocking but it does not satisfy the conditions of proposition 2. It seems, however, straightforward to generalize the proposition to recognize such cases as well.

The notions of confluence and non-blocking passive declarations can be refined when a query restriction and state invariant are given as to allow more passive declarations. The local test of proposition 2 must, then, be extended as to ensure that each step made by the operational semantics preserves the invariant; the construction seems straightforward and analogous to the one in section 4.

7 An estimation of the speedup gained by passive declarations

In order to give a mathematical characterization of the speedup gained, a very detailed operational model would be needed, far more complex than the operational semantics we use and the refined semantics that we have referred to.

Informally, we can point out the following factors for control-safe passive declarations. To observe the largest effect, assume that a large proportion of all head constraints have been made passive.

- When a constraint is called, the number of head constraints that it matches in the entire program may fall with a factor proportional with the program size (e.g., down to one).
- The work that would have been spent for finding out that each one of the unsuccessful matches did not lead to a full match may be exponential: Any combination of constraints in the store may need to be tried out.

So the conclusion is that adding passive declarations in a control-safe way may result in speedups from zero to exponential. For the answer-safe and not control-safe cases, the speedup can be arbitrarily large as any "sick" branch in the tree of possible computation paths may be pruned (but, as noted above, the speedup may also be negative in some cases).

We made some test runs for grammars in CHR in SICStus Prolog with and without passive declarations on all but the rightmost constraint in each head as described in section 5. All test were run on a Macintosh G4 1.33GHz running OS X, Version 10.3.9 and SICStus Prolog 3.12.0 and compiled with Prolog's optimizing compiler.

We used a series of highly ambiguous grammars that produce an exponential number of nodes measured in the length of the input string. We tested with 2–5 constraints in the head of each rules; the one for 3 constraints includes (with passive declarations added) these rules:

```
token(N0,N1,a) ==> a(N0,N1).
a(N0,N1)#, a(N1,N2)#, a(N2,N3) ==> a(N1,N3).
```

In addition it has 10 identical rules of the form

```
a(N0,N1)#,  a(N1,N2)#, a(N2,N3) ==> dummy.
```

We tested different input strings of length up 11 and longest running time of 45 sec's, which resulted in speedup factors of 1.5–2 with no clear correlation to the length of the input string or the number of tokens in the store.

To test the model of section 5.2, we used a program that first finds all shortest paths in a graph, and then derives some strange properties from edge and paths constraints. Surprisingly, we could observe speedups of only 0–7% for different runs. Trying to explain this, we observe that when, say, an edge constraint is called, the checking suppressed concerns exclusively rules where the constraint store at the given moment contains no constraints at all for the other predicates

mentioned in the heads of those rules. We will expect that the SICStus compiler produces code that finds out very quickly if head predicates are empty.

Finally, we did a test constructed so that the advantages of passive declarations should be obvious and where the compiler should have no chance to optimize as the gain depends heavily on the class of queries. The passive declarations in the following one-line program

```
a(X)#, b(Y)#, c(Z) ==> S is X+Y+Z, S<5 | d(X,Y,Z)
```

are control-safe for the class of queries of the form `c(1)`, ..., `c(5)`, `a(1)`, ..., `a(`n`)`, `b(1)`, ..., `b(`n`)`, `c(1)`, ..., `c(5)`. Here we observed a speedup factor proportional to n: around 10 for $n = 50$, 20 for $n = 100$, and 40 for $n = 200$, with the running times for the last example being 76 sec. versus less than 2 sec. without and with passive declarations.

8 Related work

The general outline of the compiler architecture for CHR is explained in [8], where CHR is compiled into Prolog using attributed variables. This compiler embeds the so-called refined semantics later formalized by [3], and which is necessary for an understanding of passive declarations; however, [3] do not consider passive declarations as we do with our operational semantics. As mentioned, our semantics is considerably simpler as it is intended mainly for manual proofs but it also ignores some aspects such as the order in which clauses are tried out. Adding a treatment of passive declarations to [3]'s semantics, which will be fairly straightforward, will obviously make it possible to explain correctness of more kinds of declarations in a systematic way.

In [6], the authors describe a number of optimizations made in the CHR compiler of the HAL system, most of which do not refer to passive declarations or can be explained as such. However, they explain so-called continuation optimizations where, depending on success of failure of a called constraint to trigger a rule (by gathering companion constraints to match the full head plus satisfying the guard), it is determined whether the next occurrence of that constraint predicate in a rule head needs to be checked.

Abstract interpretation is a very powerful technique for program analysis and a wide range of methods and tools are available for other types of programming languages, but as we have mentioned, CHR has a more complicated control structure than most other languages, so existing methods cannot be directly taken over. In the lack of such tools, we have made some experiments by analyzing a meta-interpreter for CHR, written in Prolog as a straightforward replication of our operational semantics. We then analyzed its behaviour for given CHR programs using a tool for abstract interpretation of Prolog [5], but it was not possible to obtain sufficiently precise information about CHR's execution state as to make interesting predictions concerning the control flow.

Very recently, [11] has proposed a framework for abstract interpretation of CHR, which we have not yet had the possibility to apply for our purpose, but this is an obvious next step.

Optimizations and analyses made by a compiler for CHR cannot take into account restrictions to classes of relevant queries as we have done, but we may refer to recent work [13, 10] that also employs such restrictions.

The present paper concerns optimizations of CHR programs which can be done at the source language level by means of passive declarations. In contrast to this, or should we say in competition with, the available CHR compilers are under continual improvement with optimizations done at lower levels which may be unpublished or difficult to trace. As we have noticed, in many cases the addition of passive declarations gives less speedup that expected which indicates that CHR compilers already do many similar optimizations, many of which go for very specific cases. We may mention a few such optimizations that we are aware of. HAL and SWI-Prolog's CHR compilers employs a "never stored" optimization [9, 6]: In case of a simplification rule of form `p(X1, ... , X`$_n$`) <=> true | ⋯`, passive declarations can be added to other rules containing `p` in the head; all constraints other than `p` in such rules can be treated as passive. This is an example of a control-safe optimization but which is obviously not caught by the local control-safety criterion we have proposed. In addition, [9] reports a method adding possible passive declarations to simplification rules of the form `p( ⋯ ) \ p( ⋯ ) <=> true | ⋯`, when a condition of functional dependency among the arguments is met. A related optimization based on so-called occurrence subsumption (also for simplification rules) is described by [12].

9 Conclusion and perspectives

We have given a framework for reasoning about passive declarations of CHR programs and indicated ways of showing their correctness in the sense that they do not change the final results. We formalized this as a notion of answer-safety, and indicated a stronger and in many cases easier to check property called control-safety.

We indicated different methods to check or identify control-safe patterns of passive declarations, and we showed how our methods can be applied to prove once and for all the correctness of strategies for adding passive declarations to particular classes of CHR programs. For programs proved (by other means) to be confluent, we have proposed a simple and easily mechanizable method for adding answer-safe passive declarations that are not necessarily control-safe.

Finally we notice that our analysis and optimization were made entirely at the high level of the source language (CHR) and a relatively abstract operational semantics. Moving our ideas down to a more detailed level of an actual compiler (cf. [6]) gives more freedom for optimizations, but we will also lose transparency in the presentation.

Acknowledgment

This work is supported by the CONTROL project, http://control.ruc.dk, funded by the Danish Natural Science Research Council. The author wants to thank

John Gallagher for helpful discussions and for cooperation on various experiments with abstract interpretation. Also thanks to anonymous reviewers of an earlier version of this paper for additional references and insightful comments on CHR compilers.

References

1. Slim Abdennadher, Thom W. Frühwirth, and Holger Meuss. On confluence of Constraint Handling Rules. In Eugene C. Freuder, editor, *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*, volume 1118 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.
2. Henning Christiansen. CHR Grammars. *Theory and Practice of Logic Programming, Special Issue on Constraint Handling Rules*, 2005. To appear.
3. Gregory J. Duck, Peter J. Stuckey, Maria J. García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2004.
4. Thom W. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programmming*, 37(1-3):95–138, 1998.
5. John P. Gallagher and Kim S. Henriksen. Abstract domains based on regular types. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 27–42. Springer, 2004.
6. Christian Holzbaur, Maria 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*, 2005. To appear.
7. Christian Holzbaur, Maria J. García de la Banda, David Jeffery, and Peter J. Stuckey. Optimizing compilation of Constraint Handling Rules. In Philippe Codognet, editor, *Logic Programming, 17th International Conference, ICLP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2237 of *Lecture Notes in Computer Science*, pages 74–89. Springer, 2001.
8. Christian Holzbaur and Thom W. Frühwirth. Compiling Constraint Handling Rules into Prolog with attributed variables. In Gopalan Nadathur, editor, *Principles and Practice of Declarative Programming, International Conference PPDP'99, Paris, France, September 29 - October 1, 1999, Proceedings*, volume 1702 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 1999.
9. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In *Proceedings of the First Workshop on Constraint Handling Rules: Selected Contributions, Ulm, Germany, May 2004*, volume 2004-01 of *Ulmer Informatik-Berichte*. Universität Ulm, Germany, 2004.
10. Tom Schrijvers and Thom W. Frühwirth. Analysing the CHR implementation of Union-Find. In Armin Wolf, Thom W. Frühwirth, and Marc Meister, editors, *W(C)LP, 19th Workshop on (Constraint) Logic Programming, Ulm, Germany, February 21-23, 2005*, volume 2005-01 of *Ulmer Informatik-Berichte*. Universität Ulm, Germany, 2005.

11. Tom Schrijvers, Peter J. Stuckey, and Gregory J. Duck. Abstract interpretation for Constraint Handling Rules. In Pedro Barahona and Amy P. Felty, editors, *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal*, pages 218–229. ACM, 2005.
12. Jon Sneyers, Tom Schrijvers, and Bart Demoen. *Guard and Continuation Optimization for Occurrence Representations of CHR.* Technical report, CW420, K.U.Leuven, Department of Computer Science, 2005.
13. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard simplification in CHR programs. In Armin Wolf, Thom W. Frühwirth, and Marc Meister, editors, *W(C)LP, 19th Workshop on (Constraint) Logic Programming, Ulm, Germany, February 21-23, 2005*, volume 2005-01 of *Ulmer Informatik-Berichte*, pages 123–134. Universität Ulm, Germany, 2005.
14. Swedish Institute of Computer Science. *SICStus Prolog user's manual, Version 3.12*, 2005. Available at `http://www.sics.se/isl`

Accurate Functional Dependency Analysis for Constraint Handling Rules

Gregory J. Duck[1] and Tom Schrijvers[2⋆]

[1] NICTA Victoria Laboratory, Department of Computer Science & Software Engineering, University of Melbourne, 3010, Australia
[2] Department of Computer Science, K.U.Leuven, Belgium

Abstract. Information about functional dependencies is used by modern CHR compilers for both optimisation and for further program analysis (e.g. confluence analysis). Before this work, CHR compilers relied on an ad hoc analysis for functional dependencies based on searching for rules of a particular form and the results from late storage analysis.
We present a more formal functional dependency analysis of CHRs based on abstract interpretation. We show, by example, that the new analysis is more accurate than the existing ad hoc analysis.

1 Introduction

The recent appearance of new optimising CHR systems [3, 5] has given rise to the need to communicate and compare between different CHR systems, which has led to the formulation of the more deterministic refined operational semantics [2] shared among CHR compilers.

Apart from the common formal semantics, there is also a need to communicate and compare program analyses. As the complexity of CHR compilers increases we need a better understanding of current analyses and ways to extend and combine them. Most of the currently existing analyses have been formulated in an ad hoc way and no formal proofs of correctness exist.

In [6] a formal framework for program analysis of CHR, based on abstract interpretation [1], has been presented. This framework provides a remedy for the current difficulties in correctly analysing CHR programs, and should enable optimising CHR compilers to reach a new level of complexity and correctness.

In this paper we apply the abstract interpretation framework of CHR to the ad hoc analysis of the functional dependency property presented in [4]. The functional dependency property is useful for program optimisations such as join ordering, index selection and never-stored optimisation. Moreover, functional dependency information allows for more accurate confluence analysis [2] with respect to the refined operational semantics.

The new functional dependency analysis is an improvement over the ad hoc version for several reasons. Firstly, it provides a formal description of the analysis, as an abstraction of the refined operational semantics. This allows for better

⋆ Research Assistant of the Fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

insight into the analysis, its correctness, possible improvements and integration with other analyses. Secondly, functional dependency information is associated with every "program point", rather than for the program as a whole. This allows for more localised optimisations. Thirdly, the analysis keeps track of constraint multiplicity information for the constraint store. This allows for more accurate approximation of the actual control flow.

Overview First, in Section 2 we summarise the operational semantics that the functional dependency property is based on. Then Section 3 formally defines the property itself and illustrates it on a number of practical examples. Next, Section 4 briefly explains the existing ad hoc analysis. In Section 5 the new analysis, its abstract domain and abstract semantics function are presented and illustrated. We discuss the implementation and evaluation of our new analysis in Section 6. Finally, Section 7 concludes.

2 Operational Semantics of Constraint Handling Rules

Functional dependency analysis relies on the refined call-based operational semantics of CHRs, which we briefly introduce here. Although this paper assumes strong familiarity with the call-based semantics of CHRs, we must omit a proper introduction for space reasons. We direct the reader to [6, 2] for an extensive treatment.

The call-based semantics for CHRs is defined as a transition system between *execution states*. An execution state is a tuple $\langle G, A, S, B, T\rangle_n$, where G is the *goal*, A is the *activation stack*, S is the *CHR store*, B is the *built-in store*, T is the *propagation history* and n the *next free constraint identifier*. Execution states are represented by the symbol σ, and $\mathbf{\Sigma}$ is the set of all execution states.

The goal G is either a *built-in constraint*, *CHR constraint*, *identified constraint* or *active constraint*. Built-in constraints are handled by an underlying built-in solver, whereas the other kinds of constraints are handled by the CHR execution algorithm. A CHR constraint c may be associated with a (unique) integer identifier i to become an *identified constraint* $c\#i$. Furthermore, $c\#i$ may be associated with an *occurrence number* j to become an *active constraint* $c\#i:j$. The identifier i distinguishes multiple copies of c, and the occurrence number j is essentially a program counter (which decides the rule to match against).

The CHR store S is a set of identified constraints. The built-in store B is an accumulated conjunction of the built-in constraints passed to the built-in solver during solving. The meaning of the built-in constraints is decided by a *constraint domain* $\mathcal{D}$. For example, if $\mathcal{D} \models B \rightarrow (x = y)$, then x and y are equal w.r.t. the constraints in B.

Operationally, a constraint c is *called* by making it the goal in an execution state. If c is a built-in constraint, then c is added to the built-in store (i.e. passed to the built-in solver). Otherwise, if c is a CHR constraint, then it is *activated* to become $c\#i:1$ (for new number i) and $c\#i$ is added to S. An active constraint will search through all occurrences (by incrementing the occurrence number)

until it is either deleted by a rule, or all occurrences and matchings have been already been tried.

3 The Functional Dependency Property

In this section we formally define the functional dependency property.

A functional dependency is a relationship between the arguments of a CHR constraint. The notation we use for functional dependencies is

$$p(x_1, \ldots, x_n) :: \{x_{i_0}, \ldots, x_{i_l}\} \rightsquigarrow \{x_{j_0}, \ldots, x_{j_m}\}$$

which indicates that the arguments $\{x_{i_0}, \ldots, x_{i_l}\}$ functionally determine the value of arguments $\{x_{j_0}, \ldots, x_{j_m}\}$, where both $\{x_{i_0}, \ldots, x_{i_l}\}$ and $\{x_{j_0}, \ldots, x_{j_m}\}$ are subsets of $\{x_1, \ldots, x_n\}$. We sometimes refer the domain $\{x_{i_0}, \ldots, x_{i_l}\}$ as the *key* for the functional dependency.

The key to detecting functional dependencies is the following utility function, which counts the number of constraints satisfying a particular form in the given CHR store.

Definition 1. *Given a functor/arity of a constraint p/n, a set of positive integers $\{i_0, \ldots, i_j\}$, CHR store S and a built-in store B, we define function $\mathsf{count}(p, n, \{i_0, \ldots, i_j\}, S, B)$ to be the following. Let $S' \subseteq S$ be the maximal subset of S such that all $p(x_1, \ldots, x_n)\#i \in S'$ and $p(y_1, \ldots, y_n)\#i' \in S'$ satisfy*

$$\mathcal{D} \models B \rightarrow (x_{i_0} = y_{i_0} \wedge \ldots \wedge x_{i_j} = y_{i_j})$$

Then $\mathsf{count}(p, n, \{i_0, \ldots, i_j\}, S, B) = |S'|$.

Example 1. For example, given the following CHR store

$$S = \{p(1,2)\#1, p(1,3)\#2, p(1,4)\#3\}$$

then $\mathsf{count}(p, 2, \{1\}, S, true) = 3$ since there is at most three constraints which share the same first argument. Similarly, $\mathsf{count}(p, 2, \{2\}, S, true) = 1$ since there is at most one constraint which share the same second argument. □

We can formally define a functional dependency in terms of the count function as follows.

Definition 2 (Set Semantic Functional Dependency). *Given a CHR store S, and built-in store B, we say constraint p/n has a* set semantic functional dependency

$$p(x_1, \ldots, x_n) :: \{x_{i_0}, \ldots, x_{i_j}\} \rightsquigarrow \{x_1, \ldots, x_n\}$$

if $\mathsf{count}(p, n, \{i_0, \ldots, i_j\}, S, B) \leq 1$.

Example 2. Consider the CHR store S from Example 1. Then the set semantic functional dependency $p(x, y) :: \{y\} \rightsquigarrow \{x, y\}$ holds since each possible value of y is associated with at most one value of $\{x, y\}$ in the store. For example, $y = 3$ is only associated with $\{1, 3\}$ from the constraint $p(1, 3)\#2$, etc.

On the other hand, the set semantic functional dependency $p(x, y) :: \{x\} \rightsquigarrow \{x, y\}$ does not hold since there exists a value for x, namely $x = 1$, which is associated with multiple values for $\{x, y\}$, i.e., $\{1, 2\}$, $\{1, 3\}$, etc. □

This definition is only concerned with *full* functional dependencies, i.e. the domain of the functional dependency determines all other arguments. A more general functional dependency only needs to determine some subset of all arguments greater than the domain, e.g., $p(x, y, z) :: \{x\} \rightsquigarrow \{x, y\}$ is a non-full functional dependency. Currently, we only analyse for full functional dependencies. Some optimisations, such as indexing [4], require full functional dependencies.

Also, our definition differs from the usual mathematical definition of a functional dependency. Strictly speaking, a functional dependency is a relationship between arguments of constraints, namely what arguments determine the values of other arguments. In a *set semantic* functional dependency, we require both a traditional functional dependency, and the requirement that there is at most one copy of a constraint with the same key in the CHR store. For example, the CHR store $\{p(1, 2)\#1, p(1, 2)\#2\}$ has a functional dependency between the first and second arguments of the `p` constraint, because given the value to the first argument we can determine the value of the second. However, there is no set semantic functional dependency, because two constraints with the same first argument simultaneously appear in the store at once. Set semantic functional dependencies are stronger than the traditional functional dependencies. CHR optimisations specifically rely on set semantic functional dependencies, hence the distinction. For brevity, we will often refer "set semantic functional dependencies" as simply "functional dependencies" from now on.

3.1 Examples of Functional Dependency

Example 3. Our first example is based on using CHRs as a queryable data structure. This is a common CHR idiom. It supports insertion and lookups of key-value ($K - V$) pairs.

```
insert(K,V) <=> entry(K,V).
entry(K,_) \ entry(K,_) <=> true.
entry(K,V) \ lookup(K,V0) <=> V0 = V.
lookup(_,_) <=> fail.
```

The second rule ensures that a functional dependency exists between the key and value for a given `entry` constraint in the store. This means that the key K determines the value V for any `entry(`K`,`V`)` appearing in the store. Essentially, if two entries exist with the same key, then one will be deleted, thus preserving the functional dependency.

The other constraints, namely `insert` and `lookup` are both *never-stored*. This is because these constraints are always deleted by either the first or fourth rule. □

The ad hoc analysis detects all functional dependencies and never-stored constraints for the database program. However there are examples of programs where such information cannot be detected by current implementations.

Example 4. The following CHR program sorts the list $[x_1, x_2, x_3, \ldots, x_n]$ represented as the CHR constraints $link(x_1, x_2), link(x_2, x_3), \ldots, link(x_{n-1}, x_n)$.

```
link(X,Y) <=> X > Y | link(Y,X).
link(X,Y) \ link(X,Z) <=> Y =< Z | link(Y,Z).
link(X,Y) \ link(T,Y) <=> X >= T | link(T,X).
```

For example, the goal $link(9,5), link(5,7), link(7,6), link(6,8)$ representing the list '$[9,5,7,6,8]$' results in $link(5,6), link(6,7), link(7,8), link(8,9)$ representing the list '$[5,6,7,8,9]$'. □

Both the second and third rule enforce functional dependencies between arguments. For example, the second rule enforces that the first argument determines the value of the second in a `link`/2 constraint.

Under current implementations of functional dependency analysis, neither functional dependency will be detected. This is because the compiler's analysis is too weak to decide the meaning of inequalities in either guard. One solution is to strengthen the existing analysis by allowing the compiler to reason about inequalities, as is done in [8] to detect redundant code. However, in some systems such as HAL there are no guarantees about the meaning of the inequality predicate (e.g. `=<`/2 could be an arbitrarily defined user predicate in HAL).

Another solution is for the programmer to add redundant rules to "declare" the functional dependency. In this case the programmer will define the following rules.

```
link(X,_) \ link(X,_) <=> true.
link(_,Y) \ link(_,Y) <=> true.
```

Note that the operational behaviour of the rules are not important, since the programmer never intends for the rules to be executed anyway. The question remains of where to insert the rules into the program such that the rules are never executed. Clearly, the rules must be inserted after the first three rules in Example 4.

Unfortunately, under the ad hoc functional dependency analysis the new rules provide no additional benefit. The reason is because the ad hoc analysis relies on the results of *late storage analysis* [4, 6]. Although rules 4 and 5 are of the appropriate form for functional dependencies, an active `link`/2 constraint will have already been stored thanks to rules 2 and 3.

Our new functional dependency analysis is independent of late storage analysis. As a result, we will show that we can still derive benefit from the addition of the new rules for this program.

4 Ad Hoc Functional Dependency Analysis

In this section we give a brief overview of the ad hoc functional dependency analysis in [4].

Essentially, the analysis relies on detecting rules of the following form in the program.

$$p(x_1, \ldots, x_n)[\backslash,]p(y_1, \ldots, y_n)_j \Longleftrightarrow x_{i_0} = y_{i_0} \wedge \ldots \wedge x_{i_m} = y_{i_m} \mid C$$

Here $x_1, \ldots, x_n, y_1, \ldots, y_n$ are distinct variables. For example, the following rule from Example 3 fits the required form.

```
entry(K,_) \ entry(K,_) <=> true.
```

Note that shared variables in the rule head indicate equation guards, e.g., the above rule is equivalent to

```
entry(K1,_) \ entry(K2,_) <=> K1 = K2 | true.
```

If such a rule exists, then generally the functional dependency $p(x_1, \ldots, x_n) :: \{x_{i_0}, \ldots, x_{i_m}\} \rightsquigarrow \{x_1, \ldots, x_n\}$ holds. To see this, assume that for a given key, e.g. $K = cat$, there already exists a constraint, say `entry(`cat`,`dog`)` in the store. If a new constraint with the same key, e.g. `entry(`cat`,`pig`)`, becomes active, then we can statically determine that the above mentioned rule will always fire, deleting the new constraint with the same key. Thus we maintain the invariant that only one constraint exists in the store per key

There is a slight complication, the refined semantics requires that the new constraint be immediately stored when it first becomes active, before any rules can fire. This means that until the rule fires, the functional dependency is being violated. Thus the ad hoc functional dependency analysis relies on *late storage analysis* [4, 6], which determines the latest point in the program where a constraint needs to be added into the store. If the rule enforcing the functional dependency appears before this point, then the functional dependency is valid, otherwise the analysis cannot infer any functional dependency.

In addition to functional dependencies, there is an ad hoc *never-stored* analysis [4], which determines if a CHR constraint is never present in the CHR store. This analysis is similar to the previous, except this time we look for rules of the from

$$p(x_1, ..., x_n) \Longleftrightarrow C$$

where $x_1, ..., x_n$ are distinct variables. If such a rule appears before an active $\mathtt{p}/n$ is stored, then we say that $\mathtt{p}/n$ is never-stored. For example, `insert` and `lookup` from Example 3 are all never-stored.

5 Accurate Functional Dependency Analysis

In this section we present the more accurate functional dependency analysis for CHRs based on the abstract interpretation framework for CHRs [6]. In particular we formulate our analysis in terms of the denotational semantics-based formulation of the framework [7].

5.1 Abstract Domain Σ_a

In this section we describe the abstract domain $\mathbf{\Sigma_a}$ for functional dependency analysis. In our abstraction we preserve information about two components: the goal and the constraint store.

The abstraction of the former retains the necessary information about the program point as required by the abstract interpretation framework. We define function α_{fd-c} that abstracts the goal component of a state.

$$\begin{array}{lll}
\alpha_{fd-c}(\Box) & = \Box & \\
\alpha_{fd-c}(c) & = \texttt{builtin} & (c \text{ built-in}) \\
\alpha_{fd-c}(p(x_1,\ldots,x_n)) & = p & \\
\alpha_{fd-c}(p(x_1,\ldots,x_n)\#i) & = p\# & \\
\alpha_{fd-c}(p(x_1,\ldots,x_n)\#i\!:\!j) & = p\!:\!j & \\
\alpha_{fd-c}([c|G]) & = [\alpha_{fd-c}(c)|\alpha_{fd-c}(G)] &
\end{array}$$

The abstraction of the latter captures patterns in the constraint store S that are essential for the analysis. Therefore, the abstract store is a set of *lookups*, which are defined by the following *lookup function*.

Definition 3 (Lookup function). *We define the* lookup function $\mathit{lookup}(p,n,K)$*, where p is a predicate symbol, n is the arity of p and $K \subseteq \{1,\ldots,n\}$ a set of integers, as follows.*

$$\begin{array}{lll}
\mathit{lookup}(p,n,K) & = p(\mathit{lookup}(1,K),\ldots,\mathit{lookup}(n,K)) & \\
\mathit{lookup}(i,K) & = * & i \in K \\
\mathit{lookup}(i,K) & = _ & i \notin K
\end{array}$$

Consider the constraint $\mathtt{p}/2$, then the set of possible lookups are

$$\{p(*,*), p(*,_), p(_,*), p(_,_)\}$$

Note that $\mathsf{lookup}(p,n,K)$ is isomorphic to K, and its usage is mainly syntactic. The concept of a *lookup* will become relevant to CHR optimisation. The set of arguments represented by the $*$s are referred to as the *key* of the lookup.

We can now define an abstraction function α_{fd-S} over the CHR store.

Definition 4. *Let S be a CHR store, and let $\mathtt{p}/n$ be the functor/arity of a CHR constraint of interest (e.g. any CHR constraint appearing in program P). Let $K \subseteq \{1,\ldots,n\}$ be a set of integers, let $c = \mathit{lookup}(p,n,K)$, let $\mathit{count} = \mathit{count}(p,n,K,S,B)$ and let $\mathit{count}' = \mathit{count}(p,n,K,S',B)$ where S' is defined as follows. Let $p(x_1,\ldots,x_n)\#i\!:\!j$ be the first active constraint in A with predicate p, then $S' = S - \{p(x_1,\ldots,x_n)\#i\}$, otherwise (if no such active constraint exists) $S' = S$. Then*

$$\begin{array}{lcl}
\mathit{count} = 0 & \Rightarrow & c^0 \in \alpha_{fd-S}(A,S,B) \\
\mathit{count} = 1 \wedge \mathit{count}' = 0 & \Rightarrow & c^{1a} \in \alpha_{fd-S}(A,S,B) \\
\mathit{count} = 1 \wedge \mathit{count}' = 1 & \Rightarrow & c^1 \in \alpha_{fd-S}(A,S,B) \\
\mathit{count} = 2 \wedge \mathit{count}' = 1 & \Rightarrow & c^{2a} \in \alpha_{fd-S}(A,S,B) \\
\mathit{count} = 2 \wedge \mathit{count}' = 2 & \Rightarrow & c^* \in \alpha_{fd-S}(A,S,B) \\
\mathit{count} \geq 2 & \Rightarrow & c^* \in \alpha_{fd-S}(A,S,B)
\end{array}$$

We define $\alpha_{fd-S}(A, S, B)$ to be the smallest possible set satisfying the above conditions.

We refer the superscript associated with each lookup as the *counter* for that lookup. The counters 0, 1 and $*$ are fairly intuitive, as they mean that there is at most 0, 1 or many constraints in the CHR store with the same key as the lookup. The special counters, $1a$ and $2a$ are slightly more complicated. The counter $1a$ is equivalent to 1 except that if we were to remove the top-most active constraint of the same functor/arity from the concrete store, then the new counter will be 0. Similarly, the counter $2a$ is equivalent to 2 (although we treat 2 the same as $*$), but if the top-most active constraint were to be removed, then the counter will be 1. We shall refer to these special counters as *marked counters*, and other counters as *unmarked counters*. Marked counters are necessary since functional dependency analysis relies on improving the counters (i.e. moving to a lower counter) if possible.

Example 5. Consider the following CHR store from the `database` program from Example 3.

$$S = \{\texttt{entry}(key, cat)\#2, \texttt{entry}(key, dog)\#1\}$$

Assume that the built-in store is trivial, i.e. $B = true$, and the given execution stack is $A = [\texttt{entry}(key, cat)\#2\texttt{:}1]$, then

$$S' = \{\texttt{entry}(key, dog)\#1\}$$

hence

$$\alpha_{fd-S}(A, S, B) = \{\texttt{entry}(*, *)^1, \texttt{entry}(*, _)^{2a}, \texttt{entry}(_, *)^1, \texttt{entry}(_, _)^{2a}\}$$

Both $\texttt{entry}(*, *)$ and $\texttt{entry}(_, *)$ have the counter 1, since there is at most one `entry` that shares the same key $\{1, 2\}$ or $\{2\}$. One the other hand, both $\texttt{entry}(*, _)$ and $\texttt{entry}(_, _)$ are have the marked counter $2a$. This is because there are two `entry` constraints that share the same key $\{1\}$ or $\emptyset$, however there would be only one such constraint if we were to remove $\texttt{entry}(key, cat)$ (the current active constraint) from consideration.

If the given activation stack had been empty, i.e. $A = []$, then $S = S'$ hence

$$\alpha_{fd-S}(A, S, B) = \{\texttt{entry}(*, *)^1, \texttt{entry}(*, _)^*, \texttt{entry}(_, *)^1, \texttt{entry}(_, _)^*\}$$

There are still 2 constraints sharing the same key $\{1\}$ or $\emptyset$. However there is no current active constraint to remove from consideration, thus the counter for $\texttt{entry}(*, _)$ and $\texttt{entry}(_, _)$ is now $*$. □

Example 6. Consider the following execution state σ for the `database` program in Example 3.

$$\langle \texttt{entry}(key, cat)\#2\texttt{:}1, [], \{\texttt{entry}(key, cat)\#2, \texttt{entry}(key, dog)\#1\} true, \emptyset \rangle_3$$

The CHR and built-in stores are the same as in Example 5. Then

$$\alpha_{fd}(\sigma) = \langle \texttt{entry:}1, \{\texttt{entry}(*, *)^1, \texttt{entry}(*, _)^{2a}, \texttt{entry}(_, *)^1, \texttt{entry}(_, _)^*\} \rangle$$

□

Finally, abstraction function α_{fd} that maps concrete states $\sigma = \langle c, A, S, B, T\rangle_n$ of the concrete domain $\mathbf{\Sigma}$ onto abstract states.

Definition 5 (Abstraction Function). *We define function α_{fd} as follows*

$$\alpha_{fd}(\langle c, A, S, B, T\rangle_n) = \langle \alpha_{fd-c}(c), \alpha_{fd-S}([c|A], S, B)\rangle$$

For any given abstract state σ we can easily determine which functional dependencies exist by interpreting the counts on the lookups. In general, a lookup with a count of 1 represents a functional dependency. For example, the lookup $\mathtt{p}(*,_)^1$ indicates the functional dependency $p(x,y) :: \{x\} \rightsquigarrow \{x,y\}$ holds for the given abstract state, and hence all corresponding program points.

The abstract interpretation framework requires a partial order to be defined over abstract states.

Definition 6 (Partial Ordering). *Let $s_0 = \langle G_0, S_0\rangle$ and $s_1 = \langle G_1, S_1\rangle$, then $s_0 \preceq_{fd} s_1$ iff $G_0 = G_1$ and $S_0 \preceq_{fd_S} S_1$. The partial order $\preceq_{fd_S}$ over abstract CHR stores is defined as follows. If for all $c^n \in S_0$, there exists a $c^m \in S_1$ (with the same c), and $c^n \preceq c^m$, then $S_0 \preceq_{fd_S} S_1$. Here we define*

$$c^0 \prec c^{1a} \prec c^1 \prec c^{2a} \prec c^*$$

if S_0 and S_1 contain different lookups, $\preceq_{fd_S}$ is undefined .[3]

We can use the definition of the partial ordering to define the concretisation function of this abstract domain as $\gamma_{fd}(s) = \{\sigma \mid \alpha_{fd}(\sigma) \preceq_{fd} s\}$. For the sake of completeness[4], we add a top element $\top_{fd}$ to the abstract domain, with $\gamma(\top_{fd}) = \mathbf{\Sigma}$ and $\forall s \in \mathbf{\Sigma_a} : s \preceq_{fd} \top_{fd}$.

Clearly the abstract domain forms a lattice with the ordering relation $\preceq_{fd}$. The least upper bound operation over abstract stores is defined as follows.

Definition 7 (Least Upper Bound). *Let $s_0 = \langle G_0, S_0\rangle$ and $s_1 = \langle G_1, S_1\rangle$, then $s_0 \sqcup_{fd} s_1 = \langle G_0, S_0 \sqcup_{fd-S} S_1\rangle$ if $G_0 = G_1$ and S_0 and S_1 contain the same lookups, otherwise it is $\top_{fd}$. The operator $\sqcup_{fd-S}$ over abstract CHR stores is defined as follows. If for all $c^n \in S_0$, there exists a $c^m \in S_1$ (with the same c), then $\mathsf{max}_{\preceq_{fd-S}}(c^n, c^m) \in S_0 \sqcup_{fd-S} S_1$. The set $S_0 \sqcup_{fd-S} S_1$ must be the minimal set satisfying the above condition. Here, function $\mathsf{max}_{\preceq_{fd-S}}$ is a maximum function using the ordering $\preceq_{fd-S}$ given in Definition 6. Otherwise $\sqcup_{fd-S}$ is undefined if S_0 and S_1 contain different lookups.*

5.2 Abstract Semantic Function $\mathcal{AS}$

The abstract semantic function $\mathcal{AS}$ for the functional dependency analysis is defined below.

[3] In the abstract interpretation, S_0 and S_1 will always have the same set of lookups.
[4] Our analysis never produces $\top_{fd}$.

Definition 8 (Abstract Semantic Function).
1. AbstractSolve

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle \texttt{builtin}, S\rangle) = \langle \Box, S_k\rangle$$

Let S_g be the maximal subset of S such that for all $p(x_1,\ldots,x_n)^c \in S_g$ we have that p is ground[5]. Let $S_{ng} = S \setminus S_g$, then

$$\begin{array}{l} S_0 = S_g \uplus \mathsf{multi}(S_{ng}) \\ s_j = \langle \Box, S_j\rangle = \sqcup_{fd}\{s_j^i \mid \mathcal{AS}[\![\mathcal{P}]\!](\langle p_i\#, S_{j-1}\rangle) = s_j^i \wedge 1 \le i \le n\}, j \ge 1 \end{array}$$

where p_i are predicates of all potentially nonground constraints. Let k be the smallest positive integer such that $s_k = s_{k-1}$.

Adding a built-in constraint, e.g. an equation to the built-in store has the potential to increase the counts of lookups for nonground constraints arbitrarily. Therefore, we use function multi on the nonground lookups, which overwrites any count by $*$.

$$\mathsf{multi}(S) = \{c^* \mid c^i \in S\}$$

In effect, we are assuming the weakest possible information for these lookups.
2. AbstractActivate

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle p, S\rangle) = \langle p{:}1, \mathsf{increase}(p, S)\rangle$$

A new constraint is added to the store, hence we must *increase* the counts for p. Here we define

$$\mathsf{increase}(p,S) = \left\{ c' \;\middle|\; c \in S \wedge \begin{array}{l}\text{if } \mathsf{functor}(c) = p \text{ then } c' = \mathsf{increase}(c) \\ \text{else } c' = c\end{array} \right\}$$

where

$$\begin{array}{lll} \mathsf{increase}(c^0) = c^{1a} & \mathsf{increase}(c^1) = c^{2a} & \mathsf{increase}(c^*) = c^* \\ \mathsf{increase}(c^{1a}) = c^{2a} & \mathsf{increase}(c^{2a}) = c^* & \end{array}$$

Notice that the resulting counts are always marked (except for $*$).
3. AbstractReactivate

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle p\#, S\rangle) = \langle p{:}1, S\rangle$$

Unlike AbstractActivate, the new active constraint is already present in the store, hence there is no need to call function increase on the abstract store.
4. AbstractDrop

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle p{:}j, S\rangle) = \langle \Box, \mathsf{unmark}(p, S)\rangle$$

There is no occurrence j for predicate p.

Because the active constraint for p no longer exists, we must *unmark* all of the counters for p.

$$\mathsf{unmark}(p,S) = \left\{ c' \;\middle|\; c \in S \wedge \begin{array}{l}\text{if } \mathsf{functor}(c) = p \text{ then } c' = \mathsf{unmark}(c) \\ \text{else } c' = c\end{array} \right\}$$

[5] This information may either be derived through groundness analysis [6] or mode declarations.

where

$$\begin{array}{lll} \mathsf{unmark}(c^0) = c^0 & \mathsf{unmark}(c^1) = c^1 & \mathsf{unmark}(c^*) = c^* \\ \mathsf{unmark}(c^{1a}) = c^1 & \mathsf{unmark}(c^{2a}) = c^* & \end{array}$$

In effect, the (former) active constraint is now treated the same as any other constraint in the store.

5. AbstractSimplify

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle p\!:\!j, S\rangle) = s$$

Let r be the rule which contains the jth occurrence of predicate p. If we assume that the rule fired, then

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle \alpha_{fd-c}(C), \mathsf{decrease}(p, S)\rangle) = \langle \Box, S_1\rangle = s_1$$

Because the active constraint has been deleted, we must *decrease* all of the counters for p.

$$\mathsf{decrease}(p, S) = \left\{ c' \;\middle|\; c \in S \wedge \begin{array}{l} \text{if } \mathsf{functor}(c) = p \text{ then } c' = \mathsf{decrease}(c) \\ \text{else } c' = c \end{array} \right\}$$

where

$$\begin{array}{lll} \mathsf{decrease}(c^0) = c^0 & \mathsf{decrease}(c^1) = c^1 & \mathsf{decrease}(c^*) = c^* \\ \mathsf{decrease}(c^{1a}) = c^0 & \mathsf{decrease}(c^{2a}) = c^1 & \end{array}$$

Note that we can only alter the *marked* counters. Also, the resulting counts are unmarked, since the active constraint has been deleted.

We consider the following three cases for deriving the resulting state s.

1. If rule r is of the form

   $$p(x_1, \ldots, x_n)[\backslash,]p(y_1, \ldots, y_n)_j \Longleftrightarrow x_{i_0} = y_{i_0} \wedge \ldots \wedge x_{i_m} = y_{i_m} \mid C$$

   (where occurrence j is shown).
   If we assume the rule did not fire, then the resulting state is

   $$\mathcal{AS}[\![\mathcal{P}]\!](\langle p\!:\!(j+1), \mathsf{decrease_2a}(p, n, \{i_0, \ldots, i_m\}, S)\rangle) = s_2$$

   We use function decrease_2a to improve the counts of lookups which contain key $\{i_0, \ldots, i_m\}$, where

   $$\mathsf{decrease_2a}(p, n, K, S) = \left\{ c' \;\middle|\; \begin{array}{l} c \in S \wedge \\ \text{if } \exists K' \supseteq K \text{ such that } c = \mathsf{lookup}(p, n, K') \\ \quad \text{then } c' = \mathsf{decrease_2a}(c) \\ \text{else } c' = c \end{array} \right\}$$

   and

   $$\begin{array}{lll} \mathsf{decrease_2a}(c^0) = c^0 & \mathsf{decrease_2a}(c^1) = c^1 & \mathsf{decrease_2a}(c^*) = c^* \\ \mathsf{decrease_2a}(c^{1a}) = c^{1a} & \mathsf{decrease_2a}(c^{2a}) = c^1 & \end{array}$$

We can make this improvement since if c^{2a} were in the abstract store for some lookup c with a key containing $\{i_0, \ldots, i_m\}$, then the rule *must* have fired. This kind of improvement is the essential part of functional dependency analysis.

For the resulting state we have $s = s_1 \sqcup_{fd} s_2$.

2. If rule r is an unconditional simplification rule (i.e. the guard is *true*) of the form

$$p(x_1, \ldots, x_n)_j \Longleftrightarrow C$$

then $s = s_1$.

3. Otherwise (r is not in any of the above forms), then

$$s = s_1 \sqcup_{fd} \mathcal{AS}[\![\mathcal{P}]\!](\langle p{:}(j+1), S\rangle)$$

6. AbstractPropagate

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle p{:}j, S\rangle) = \mathcal{AS}[\![\mathcal{P}]\!](\langle p{:}(j+1), S_k\rangle)$$

1. If the following conditions hold:
   - Rule r is of the form

$$p(x_1, \ldots, x_n)_j[\backslash,]p(y_1, \ldots, y_n) \Longleftrightarrow x_{i_0} = y_{i_0} \wedge \ldots \wedge x_{i_m} = y_{i_m} \wedge g \mid C$$

     (where occurrence j is shown). Here g represents an arbitrary guard.[6]
   - For $c = lookup(p, n, \{i_0, ..., i_m\})$ we have that $c^{2a} \in S$.

   Then

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle \alpha_{fd-c}(C), \mathsf{decrease_2a}(p, n, \{i_0, \ldots, i_m\}, S)\rangle) = \langle \Box, S_k\rangle$$

   defines S_k.
2. Otherwise, let

$$S_0 = S$$
$$s_j = \langle \Box, S_j\rangle = \mathcal{AS}[\![\mathcal{P}]\!](\langle \alpha_{fd-c}(C), S_{j-1}\rangle)$$

   and let k be the smallest positive integer such that $s_k = s_{k-1}$.

7. AbstractGoal

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle \Box, S\rangle) = \langle \Box, S'\rangle$$
$$\mathcal{AS}[\![\mathcal{P}]\!](\langle [c|G], S\rangle) = \mathcal{AS}[\![\mathcal{P}]\!](\langle G, S'\rangle)$$

where

$$\mathcal{AS}[\![\mathcal{P}]\!](\langle c, S\rangle) = \langle \Box, S'\rangle$$

[6] Ideally g contains (nor entails) no equations of the from $x_k = y_k$ for $1 \leq k \leq n$. If g does entail such an equation, then the analysis will be weaker than it could be.

5.3 Example Analysis

Consider the following rule on the `entry` constraint from Example 3.

```
entry(K,_) \ entry(K,_) <=> true.
```

The abstract derivations for executing a single `entry` constraint are shown in Figure 1. For brevity, we have abbreviated `entry` to `e`, and we have omitted the subcomputations for AbstractPropagate. In each instance, the AbstractPropagate does not change the abstract store. We are assuming that the `entry` constraint is a ground constraint.

After three iterations we arrive at a fixed point for the final state. The resulting abstract store is

$$\{\texttt{entry}(*,*)^1, \texttt{entry}(*,_)^1, \texttt{entry}(_,*)^*, \texttt{entry}(_,*)^*\}$$

This indicates that after an `entry` constraint has finished being active, the set semantic functional dependencies $\texttt{entry}(X,Y) :: \{X\} \rightsquigarrow \{X,Y\}$ and $\texttt{entry}(X,Y) :: \{X,Y\} \rightsquigarrow \{X,Y\}$ hold.

6 Implementation and Evaluation

Information about functional dependencies and never-stored is used in further program analysis (e.g. confluence analysis [2]) and for join ordering, index selection and never-stored optimisations. We have implemented a simple prototype functional dependency analysis in SWI-Prolog, and use the results of the analysis to improve the compilation of HAL CHR programs. Note that currently the analysis is not integrated in the HAL compiler itself, so (in some cases) the optimisations have been performed by hand. Full integration remains future work.

In Table 1 we compare several CHR programs with/without functional dependency (and never-stored) optimisations. The source code and description of the programs can be downloaded from `http://www.cs.mu.oz.au/~gjd/chr05/`. The $-fd$ ($+fd$) version has the optimisation disabled (enabled). Overall we see a 17% improvement for the $+fd$ version, which shows that functional dependency analysis is beneficial.

For most of the programs the ad hoc analysis gives equivalent results to the new analysis. The exceptions are the `sort` example (as explained in Section 3.1) and the `game` program, where one of the CHR constraints is locally never-stored for some occurrences (rather than all occurrences as required by the ad hoc analysis). In both of these programs, the ad hoc analysis is equivalent to the $-fd$ no analysis version. This shows that the new analysis is an improvement over the ad hoc version.

7 Conclusion

In this paper we have replaced the old ad hoc functional dependency analysis of [4] with a more powerful analysis. We have provided a full formal specification of

$$
\begin{aligned}
&\mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}, \{\mathsf{e}(*,*)^0, \mathsf{e}(*,_)^0, \mathsf{e}(_,*)^0, \mathsf{e}(_,_)^0\}\rangle)\\
&\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}:1, \{\mathsf{e}(*,*)^{1a}, \mathsf{e}(*,_)^{1a}, \mathsf{e}(_,*)^{1a}\mathsf{e}(_,_)^{1a}\}\rangle) && (\textbf{Activate})\\
&\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \square, \{\mathsf{e}(*,*)^0, \mathsf{e}(*,_)^0, \mathsf{e}(_,*)^0\mathsf{e}(_,_)^0\}\rangle)\\
&\qquad \sqcup_{fd} \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}:2, \{\mathsf{e}(*,*)^{1a}, \mathsf{e}(*,_)^{1a}, \mathsf{e}(_,*)^{1a}\mathsf{e}(_,_)^{1a}\}\rangle) && (\textbf{Simplify})\\
&\quad = \langle \square, \{\mathsf{e}(*,*)^0, \mathsf{e}(*,_)^0, \mathsf{e}(_,*)^0\mathsf{e}(_,_)^0\}\rangle \sqcup_{fd} \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^1\mathsf{e}(_,_)^1\}\rangle\\
&\quad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^1\mathsf{e}(_,_)^1\}\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}:2, \{\mathsf{e}(*,*)^{1a}, \mathsf{e}(*,_)^{1a}, \mathsf{e}(_,*)^{1a}\mathsf{e}(_,_)^{1a}\}\rangle)\\
&\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}:3, \{\mathsf{e}(*,*)^{1a}, \mathsf{e}(*,_)^{1a}, \mathsf{e}(_,*)^{1a}\mathsf{e}(_,_)^{1a}\}\rangle) && (\textbf{Propagate})\\
&\quad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^1\mathsf{e}(_,_)^1\}\rangle && (\textbf{Drop})
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^1, \mathsf{e}(_,_)^1\}\rangle)\\
&\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}:1, \{\mathsf{e}(*,*)^{2a}, \mathsf{e}(*,_)^{2a}, \mathsf{e}(_,*)^{2a}\mathsf{e}(_,_)^{2a}\}\rangle) && (\textbf{Activate})\\
&\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^1\mathsf{e}(_,_)^1\}\rangle)\\
&\qquad \sqcup_{fd} \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}:2, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^{2a}\mathsf{e}(_,_)^{2a}\}\rangle) && (\textbf{Simplify})\\
&\quad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^1\mathsf{e}(_,_)^1\}\rangle \sqcup_{fd} \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^*\mathsf{e}(_,_)^*\}\rangle\\
&\quad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^*\mathsf{e}(_,_)^*\}\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}:2, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^{2a}\mathsf{e}(_,_)^{2a}\}\rangle)\\
&\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}:3, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^{2a}\mathsf{e}(_,_)^{2a}\}\rangle) && (\textbf{Propagate})\\
&\quad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^*\mathsf{e}(_,_)^*\}\rangle && (\textbf{Drop})
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^*, \mathsf{e}(_,_)^*\}\rangle)\\
&\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}:1, \{\mathsf{e}(*,*)^{2a}, \mathsf{e}(*,_)^{2a}, \mathsf{e}(_,*)^*\mathsf{e}(_,_)^*\}\rangle) && (\textbf{Activate})\\
&\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^*\mathsf{e}(_,_)^*\}\rangle)\\
&\qquad \sqcup_{fd} \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}:2, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^*\mathsf{e}(_,_)^*\}\rangle) && (\textbf{Simplify})\\
&\quad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^*\mathsf{e}(_,_)^*\}\rangle \sqcup_{fd} \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^*\mathsf{e}(_,_)^*\}\rangle\\
&\quad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^*\mathsf{e}(_,_)^*\}\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}:2, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^*\mathsf{e}(_,_)^*\}\rangle)\\
&\quad = \mathcal{AS}[\![\mathcal{P}]\!](\langle \mathsf{e}:3, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^*\mathsf{e}(_,_)^*\}\rangle) && (\textbf{Propagate})\\
&\quad = \langle \square, \{\mathsf{e}(*,*)^1, \mathsf{e}(*,_)^1, \mathsf{e}(_,*)^*\mathsf{e}(_,_)^*\}\rangle && (\textbf{Drop})
\end{aligned}
$$

Fig. 1. Example abstract execution

Table 1. Timings (ms) showing the benefit of functional dependency based optimisations

Program	$-fd$	$+fd$
`database(100000,5)`	1694	1518
`union(160)`	1583	853
`cycle(14)`	664	595
`queue(100000)`	1204	1006
`queens(15)`	1117	914
`game(30000)`	1887	1761
`sort(100000)`	2624	2486
geom. mean	1424	83%

our analysis, in terms of an instantiation of the abstract interpretation framework for CHR of [6]. Even though the formulation of our analysis is relatively simple, evaluation shows that fairly strong results are obtained.

7.1 Future Work

In future work we would like to investigate a number of improvements of the functional dependency analysis, on account of both its efficiency and strength. Firstly, we would like to experiment with more compact representations of the abstract domain. If $\mathsf{lookup}(p,n,K)^i \in S$ with S an abstract constraint store, then $\forall K \subset K' : \mathsf{lookup}(p,n,K')^j \in S \wedge j \leq i$. If $i = j$ for some K, K', then we could simply omit $\mathsf{lookup}(p,n,K')^j$ from S. When looking for the counter of a K', we take $\min_{K \subset K'} \mathsf{lookup}(p,n,K)$.

Secondly, there is some information in the abstract domain that is currently not exploited in the abstract transition function. Namely, if the counter of a particular $\mathsf{lookup}(p,n,K)^j$ is smaller than the number of copies of a constraints p/n in a CHR rule, then this rule cannot be applied. This provides richer control flow information.

Thirdly, the approach of reasoning about guards, as done in [8], could prove a useful improvement to the current analysis domain.

In addition we would also like to combine the functional dependency analysis with other analyses formulated in terms of the abstract interpretation framework. In particular, the late storage analysis of [6] provides relevant information concerning the storage of the active constraint. The other way around, the functional dependency abstract domain provides useful information about the constraint store and the number of candidates for the active constraint.

References

1. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unifed Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In

POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 238–252, Los Angeles, California, 1977. ACM Press.

2. G. Duck, P. Stuckey, M. Garcia de la Banda, and C. Holzbaur. The refined operational semantics of constraint handling rules. In *20th International Conference on Logic Programming (ICLP'04)*, pages 90–104, Saint-Malo, France, September 2004.
3. C. Holzbaur, M. Garcia de la Banda, P. Stuckey, and G. Duck. Optimizing Compilation of Constraint Handling Rules in HAL. *Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules*, 2005. To appear.
4. C. Holzbaur, P. Stuckey, M. Garcia de la Banda, and D. Jeffery. Optimizing compilation of constraint handling rules. In P. Codognet, editor, *Logic Programming: Proceedings of the 17th International Conference*, LNCS, pages 74–89. Springer-Verlag, 2001.
5. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: Implementation and application. In *First workshop on constraint handling rules: selected contributions*, 2004. Published as technical report: Ulmer Informatik-Berichte Nr. 2004-01, ISSN 0939-5091, http://www/informatik.uni-ulm.de/epin/pw/10481.
6. T. Schrijvers, P. Stuckey, and G. Duck. Abstract Interpretation for Constraint Handling Rules. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2005. (to appear).
7. Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, June 2005.
8. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard reasoning for CHR optimization. Report CW 411, Department of Computer Science, K.U.Leuven, Leuven, Belgium, May 2005.