Relating Constraint Handling Rules to Datalog

Beata Sarna-Starosta[1], David Zook[2], Emir Pasalic[2], and Molham Aref[2]

[1] Department of Computer Science & Engineering
Michigan State University, East Lansing, MI 48824,
Logic Blox, Atlanta, GA 30309
E-mail: bss@cse.msu.edu
[2] Logic Blox, Atlanta, GA 30309
E-mail: {david.zook,emir.pasalic,molham.aref}@logicblox.com

Abstract. Datalog$^{\mathrm{LB}}$ is an extension of Datalog supporting global stratification of negation and functional dependencies, designed for use in industrial-scale decision automation applications. Constraint Handling Rules (CHR) is a declarative rule-based programming language, particularly suitable for specifying custom constraint solvers at a high level. Our goal is to enhance Datalog$^{\mathrm{LB}}$ with CHR-like capabilities in order to improve its expressive power and open it to specification of general-purpose constraint solvers for industrial applications. In this paper we relate the two formalisms and define a translation of a significant class of CHR programs into Datalog$^{\mathrm{LB}}$. It turns out that the translation enables reasoning about the properties of CHR programs at a high level of Datalog logic.

1 Introduction

Constraint Handling Rules (CHR) [1] is a declarative formalism of multi-headed, committed-choice, guarded, multiset rewriting rules, originally designed for extending host languages, such as Prolog or Haskell, with user-defined constraint solvers. The expressiveness of CHR facilitates specification of a wide variety of problems at a high level, and its clean semantics supports program analysis and transformation, enabling non-trivial performance improvements. Thus, the formalism has evolved into a general-purpose programming language, with application domains including computational linguistics, software engineering, deductive databases, semantic web, and more.

We consider CHR in the context of its potential benefits for the domain of automated decision support. We are developing a framework for building configurable decision support systems, based on a reasoning language rooted in Datalog. Although more powerful than pure Datalog, our language—called Datalog$^{\mathrm{LB}}$—often lacks the flexibility and generality to define constraint solvers for industrial applications. Thus, we aim to port some of the features of CHR to Datalog$^{\mathrm{LB}}$ in order to improve the latter's expressive power, and to make our systems capable of handling more complex problems.

This paper reports our experience from the initial step towards porting the functionality of CHR to Datalog$^{\mathrm{LB}}$, in which we establish a relationship between the two formalisms. The benefit of this step is mutual. On one hand, by exposing aspects of CHR that exceed Datalog$^{\mathrm{LB}}$'s handling capabilities, it fosters better design decisions leading to the improvement of our language. On the other hand, by demarcating the classes of

CHR programs that map into pure Datalog and Datalog$^{\mathrm{LB}}$, it allows reasoning about the properties of these program classes at the level of the declarative and well-studied Datalog logic. In summary, our paper makes the following contributions:

- it formally defines a basic translation schema from CHR to pure Datalog
- it identifies a class of CHR programs that are amenable for the basic translation, and thus share properties with pure-Datalog programs
- it extends the basic translation schema to accommodate properties of CHR not compliant with pure Datalog, but expressible in Datalog$^{\mathrm{LB}}$
- it identifies a class of CHR programs that are amenable for the extended translation, and thus share properties with Datalog$^{\mathrm{LB}}$ programs.

We illustrate different stages of our translation with examples of CHR programs and their pure-Datalog or Datalog$^{\mathrm{LB}}$ counterparts. All CHR programs are either borrowed or adapted from the WebCHR online demo [2].

In the remainder of the paper, Section 2 provides background on Datalog, Datalog$^{\mathrm{LB}}$ and CHR; Section 3 defines the basic translation schema from a restricted subclass of CHR to pure Datalog; Section 4 extends the basic translation schema onto a larger class of CHR and Datalog$^{\mathrm{LB}}$; Section 5 considers some of the challenges of the unrestricted CHR translation; and Section 6 concludes with a discussion and review of related work.

2 Preliminaries

We use standard notions of variables, constants, n-ary functions, terms and clauses [3], and of databases and the relational algebra [4]. We use, possibly subscripted, upper-case letters to denote sets and sequences of entities, and lower-case letters to denote their elements. Usually, t refers to a term in general, c refers to a constraint term, with a constraint symbol at the root, and d refers to a Datalog clause. We use $\mathit{vars}(t)$ to denote the set of variables in a term t. We extend the functions defined over elements of the collections onto the entire collections whenever needed and obvious from the context.

2.1 Datalog

Datalog is a subset of Prolog developed in 1970s for deductive databases. The original specification of Datalog, which we call *pure Datalog*, extended traditional database query languages with support for recursion, at the same time avoiding Prolog's non-termination issues, and thus conforming to the set-at-a-time reasoning scheme of the relational algebra. Syntactically, pure Datalog coincides with Prolog restricted so that: (i) unit clauses (facts) are always ground, (ii) all predicate arguments are variables or constants, and (iii) it is negation-free. To ensure that the condition (i) is satisfied, all clauses must be *safe*, i.e., every variable in a clause must appear in the clause's body.

The specification of pure Datalog is too confining for many practical applications, and numerous extensions have been proposed to relax its restrictions. Most notably, these include different models for handling negation, admitting disjunction in clause heads, and support for object-oriented programming. In recent years, Datalog received

attention of researchers from areas such as program analysis [5], networks [6], security protocols [7], knowledge representation [8], robotics [9], games [10], and more.

The power of Datalog lies in its ability to define new predicates, which are calculated in terms of given data. A Datalog program operates on two disjoint sets of predicates: the *extensional database (EDB)*—the predicates defined by externally supplied facts—and the *intensional database (IDB)*—the predicates calculated based on the EDB and the program rules. Semantically, a model of a Datalog program is a choice of IDB relations that, with the given EDB relations, makes all program rules true for all variable substitutions. Like Prolog, pure Datalog features *set semantics*, meaning that it does not distinguish between multiple instances of the same fact.

Datalog$^{\mathrm{LB}}$. The goal of Datalog$^{\mathrm{LB}}$ is to provide a generally useful, declarative way for expressing data structures, relations between data entities, sophisticated calculations, integrity constraints, and transactional processing. Datalog$^{\mathrm{LB}}$ is a type-safe variant of Datalog, based on incremental evaluation, with trigger-like functionality and support for dynamic updates, ability to declaratively specify functional dependencies, non-deterministic choice, stratified negation and aggregation, and meta-programming. Datalog$^{\mathrm{LB}}$ retains pure Datalog's set semantics, admitting at most one instance of any fact in program's database.

2.2 Constraint Handling Rules

Syntax. A CHR program is a finite set of rules that specify how multisets of *user-defined* constraints are solved based on the host language's *built-in* constraints (e.g. Prolog predicates). CHR rules are of the form:

$$\textit{label} \ @ \ \textit{Head} \left\{ \begin{array}{l} <=> \\ ==> \end{array} \right\} \textit{Guard} \mid \textit{Body}$$

The most general are *simpagation* rules of the form $H_1 \setminus H_2$ <=> $G \mid B$ where H_1 and H_2 are sequences of user-defined constraint terms (the *heads* of the rule), G (the *guard*) is a sequence of built-in constraints and B (the *body*) is a sequence of built-in and user-defined constraint terms. A rule specifies that when constraints in the store match H_1 and H_2 and the guard G holds, the constraints that match H_2 can be *replaced* by the corresponding constraints in B. The literal `true` represents an empty sequence of constraint terms. The guard part, $G \mid$, may be omitted when $G = \textit{true}$.

A *simplification* rule, which has the form H_2 <=> $G \mid B$ can be represented by a simpagation rule `true` $\setminus H_2$ <=> $G \mid B$. Similarly, a *propagation* rule, which has the form H_1 ==> $G \mid B$, can be represented by a simpagation rule $H_1 \setminus$ `true` <=> $G \mid B$.

Semantics. CHR has a well-defined declarative as well as operational semantics [1, 11]. The declarative interpretation of a CHR program P is given by the set of universally quantified formulas corresponding to the CHR rules, and an underlying consistent constraint theory.The constraint theory defines the meaning of host language constraints, the equality constraint '=', and the boolean atoms *true* and *false*.

The original operational semantics of CHR [11] is given in terms of a non-deterministic transition system. The evaluation of a program P is a path through the transition system. The transitions are made when a constraint is added from the goal to the store,

```
:- constraints parent/2, ancestor/2, sibling/2.

parent   @ parent(X,Y) ==> ancestor(X,Y).
ancestor @ parent(X,Y), ancestor(Y,Z) ==> ancestor(X,Z).

sibling  @ parent(P,X), parent(P,Y) ==> X\==Y | sibling(X,Y).

setsem   @ sibling(X,Y) \ sibling(X,Y) <=> true.
```

Table 1. A CHR program for deductive database of family relations

or by firing any applicable program rule. The refined operational semantics [12], followed by most CHR implementations, defines a more deterministic transition system specifying, among others, the order in which rules are tried. The refined operational semantics is shown to be sound and to have better termination behavior than the original semantics.

Example 1. Table 1 lists a CHR program encoding a simple deductive database of family relations. The propagation rule `parent`, for each stored constraint matching `parent(X,Y)`, adds to the store an `ancestor(X,Y)` constraint; the propagation rule `ancestor`, for each stored pair of constraints matching `parent(X,Y)` and `ancestor(Y,Z)`, adds an `ancestor(X,Z)` constraint; similarly, the propagation rule `sibling`, for each pair of stored constraints that match `parent(P,X)` and `parent(P,Y)`, where X≠Y, adds a `sibling(X,Y)` entry to the constraint store. The simpagation rule `setsem`, in the presence of two identical stored constraints matching `sibling(X,Y)`, replaces one of these constraints (to the right of '\') with `true`, effectively removing the constraint from the store.

The evaluation of the program is triggered by a goal formed by a sequence of user-defined constraints. The propagation rules deduce the `ancestor` and `sibling` constraints implied by the given `parent` constraints, whereas the simpagation rule `setsem` removes duplicate occurrences of the `sibling` constraint.

3 Basic Translation Schema: CHR to Pure Datalog

In this section we analyze Constraint Handling Rules in the context of the characteristic properties of pure Datalog, and identify the class of CHR directly expressible as pure-Datalog programs.

3.1 CHR vs. pure Datalog

Always-ground facts. Evaluation of a query over a pure-Datalog program computes a fixed point according to a set of rules and a relational database expressed as a set of facts, and the groundness of the facts guarantees its termination. In the context of CHR, facts correspond to the contents of the constraint store. To ensure that all constraints stored

during the evaluation of a CHR program are ground, we require that (i) the program is *range restricted*, meaning that whenever a variable appears in a program rule's body, it also appears in the rule's head[1], and (ii) all queries issued to the program are ground.

No function symbols. Pure Datalog's restriction to 0-ary function symbols manifests in the context of CHR as two requirements: (i) that all constraint arguments in program rules are variables or constants, and (ii) that no functions (e.g., arithmetic) are evaluated in program rules' bodies. For programs in which function symbols appear as arguments only in the rules' heads, we can lift the requirement (i) by applying one of the flattening techniques introduced in [13]. Thus, this restriction admits to translation into pure Datalog all CHR programs as long as their flattened versions comply with all other requirements discussed in this section.

Negation freedom. Datalog's property of negation freedom allows adding new facts to a database, but does not allow removing any facts that are already there. Thus, at any step of the evaluation, new facts are derived based on all facts added in the previous steps. Furthermore, since we only add facts, and do so until no more facts are implied by the current facts set and program rules, the order in which the facts are derived (i.e., the order in which the program rules are applied) does not affect the final result of the evaluation, meaning that all pure-Datalog programs are confluent. On the other hand, CHR supports constraint removal by means of simplification, which enables writing non-confluent programs. In this section, when relating CHR to pure Datalog, we consider only the simplification-free subset CHR. In Section 4 we identify a class of programs with restricted simplification, which can be represented in Datalog$^{\mathrm{LB}}$, and in Section 5 we discuss issues around mapping to Datalog CHR with full-fledged simplification.

3.2 Translation schema

We now characterize the class of CHR programs amenable for direct translation into pure Datalog, and define the translation schema for this program class.

Definition 1 (CHR$^{\delta}$ rule). *A* CHR$^{\delta}$ rule *is a range-restricted CHR propagation rule, in which all arguments of the body constraints are terms of arity* < 1.

Definition 2 (CHR$^{\delta}$ rule mapping). *The mapping* $m_{\delta} : CHR^{\delta} \mapsto D$ *from CHR*$^{\delta}$ *rules to pure-Datalog clauses is defined as:*

$$m_{\delta}(\texttt{H ==> G | B}) \quad = \quad \texttt{B <- H,G.}$$

Because the semantics of Datalog does not distinguish duplicate facts, translating into Datalog CHR programs that place multiple instances of the same constraint in the store will change their behavior. In our previous work [14] we identified the class of *set-CHR* programs, for which the constraint store is always a set. Clearly, translation to Datalog is useful only for set-CHR programs. A common way to enforce set semantics in CHR is by enhancing the programs with simpagation rules of the form: `c \ c <=> true.` for every constraint symbol `c`, for which multiple constraint instances may be added to the store during the evaluation. Rules of this kind, which we

[1] this property coincides with the safety property of Datalog rules (see Section 2.1)

call *set-semantics rules*, explicitly remove duplicate constraints, and are often utilized in set-CHR programs. Guaranteed semantics of the output of our translation allows to omit the set-semantics rules from the input programs:

Definition 3 (Set-semantics rule elimination). *The elimination of CHR set-semantics rules m_σ : CHR$^\delta \mapsto D$ is defined as:*

$$m_\sigma(\texttt{c} \setminus \texttt{c} \texttt{ <=> } \texttt{true}) \quad = \quad \textit{true}.$$

Recall that the set of predicates in a Datalog program is partitioned into the EDB and the IDB. Intuitively, EDB predicates are editable by the outside world and cannot be modified by the system, whereas IDB predicates are calculated by the system and cannot be edited by the outside world. In the context of CHR, the EDB are the constraints provided by queries, and the IDB are the constraints deduced by rule application.

Example 2. In the family database program in Table 1, the constraint symbol `parent` appears in the heads of all rules, and never in rule bodies. Thus, all instances of the constraint in the constraint store are those provided by the goal. Furthermore, `parent` is a premise for deducing all other constraints, as the presence of its instances in the store warrants applicability of all propagation rules. Clearly, this constraint represents an EDB predicate. By contrast, the constraint symbols `ancestor` and `sibling` appear in rule bodies, and so, the instances of these constraints are deduced by rule application. As such, the constraints represent the IDB predicates.

Even with a clean distinction between the constraint representation of the EDB and IDB in a CHR program, as in Example 2, nothing prevents posing the constraints representing the IDB in the queries, thus confusing these constraints with those representing the EDB. To avoid similar confusion in Datalog programs generated by our translation, we explicitly separate the constraints representing the IDB in program rules from their counterparts allowed in the queries by introducing an *EDB predicate* and an *EDB rule* for each constraint representing an IDB predicate in source CHR programs:

Definition 4 (EDB predicate and EDB rule). *The* EDB predicate p_ϵ *represents the IDB predicate p in the EDB. The* EDB rule *for a predicate p, $r_\epsilon(p)$, maps the EDB predicate p_ϵ to its IDB counterpart:*

$$r_\epsilon(p) \quad = \quad p \texttt{ <- } p_\epsilon$$

Definition 5 (Program translation). *A pure-Datalog translation of a CHR program given by a set of CHR$^\delta$ rules and a set of set-semantics rules over a set of user-defined constraints, $P(C) = R_\delta \cup R_\sigma$, is a program $m_\pi(P)$ in which each constraint $c \in C$ representing an IDB predicate is associated with an EDB rule, each rule $r \in R_\delta$ is mapped to a pure-Datalog clause, and all set-semantics rules are eliminated: $m_\pi(P) = r_\epsilon(C) \cup m_\delta(R_\delta) \cup m_\sigma(R_\sigma)$.*

Example 3. The family database program in Table 1 consists of three CHR$^\delta$ rules (`parent`, `ancestor`, and `sibling`), and a set-semantics rule (`setsem`). The constraint `parent` represents the EDB predicate, whereas the constraints `ancestor` and `sibling` represent the IDB predicates. The pure-Datalog translation of the program is shown in Table 2, where lines $1, 2$ list the EDB rules, lines $4, 5, 7$ list direct mapping of the CHR$^\delta$ rules to pure Datalog, and the set-semantics rule has been eliminated.

```
ancestor(X,Y) <- ancestor_ε(X,Y).                                    1
sibling(X,Y) <- sibling_ε(X,Y).                                      2
                                                                     3
ancestor(X,Y) <- parent(X,Y).                                        4
ancestor(X,Z) <- parent(X,Y), ancestor(Y,Z).                         5
                                                                     6
sibling(X,Y) <- parent(P,X), parent(P,Y), X\=Y.                      7
```

Table 2. A pure-Datalog representation of the family database program

Correctness The CHR programs expressible in pure Datalog are confluent and their evaluation always terminates. The basic translation presented in this section is sound and complete w.r.t. the data sets deduced by the input and output programs.

Theorem 1 (Soundness and Completeness). *The constraint store resulting from the evaluation of a goal Q over a CHR program $P = R_\delta \cup R_\sigma$ is equivalent to the set of facts deduced by the pure-Datalog translation of P, $m_\pi(P)$, for a set of EDB facts F such that $F = Q$.*

Theorem 1 holds based on the pure-Datalog semantics and the logical reading of CHR.

4 Extended Translation Schema: CHR to Datalog$^{\mathrm{LB}}$

In Section 3.2 we defined a mapping from CHR to pure Datalog. The mapping is straightforward, and the subclass of CHR programs amenable for the mapping are guaranteed to have the properties of pure Datalog programs such as termination and confluence. This subclass of CHR, however, is very small and leaves out many practical programs. In this section we propose three extensions to the basic translation schema, which accommodate features common in CHR, but not standard to pure Datalog. The extensions, facilitated by the properties of the Datalog$^{\mathrm{LB}}$ system underlying our translation, and by a simple CHR program transformation, still yield a translation schema that guarantees well-behavedness of the input programs.

4.1 Restricted simplification

Pure Datalog's requirement of negation freedom is perhaps the most prohibitive restriction of the language, and numerous approaches have been taken towards relaxing it. For example, many Datalog systems allow programs with *stratified negation*, i.e., programs in which all instances of any predicate appearing in negated subgoals are computed before the predicate is used with negation.

The negation-freedom requirement is very restrictive also in the context of our translation. As argued in Section 3.1, CHR implements negation by means of simplification. Hence, the requirement bans all simplification from the basic translation schema, which severely limits the schema's applicability, as simplification rules are dominant in most

CHR programs. In this section we identify a subclass of CHR with *stratified simplification*, for which we can lift this restriction.

Intuitively, a program is simplification stratified if it is never the case that the simplification of a constraint triggers further propagation. Our formal definition of simplification-stratified programs is based on the notion of *constraint dependency graph*:

Definition 6 (Constraint dependency graph). *A* constraint dependency graph *for a CHR program P is a graph $G = \langle N, E \rangle$ with a set of nodes N and a set of edges E, in which the set of nodes is the set of user-defined constraints, and there is an edge on a rule $r \in P$, denoted $e(r)$, from a source node c_s to a target node c_t, if the constraint represented by c_s appears in the head of the rule r, and the constraint represented by c_t appears in the body of r.*

Definition 7 (Negative edge and positive edge). *A* negative edge *in a constraint dependency graph is an edge $e(r)$ from a source node c_s such that r is a simplification rule, or r is a simpagation rule and its application removes the constraint c_s. A* positive edge *is an edge that is not negative.*

Definition 8 (Simplification-stratified program). *A* simplification stratified program *is a CHR program such that for all nodes in its constraint dependency graph it holds that if a node has an outgoing edge, then all its incoming edges are positive.*

The evaluation of a simplification-stratified program can be split into two conceptual steps, with a propagation step—computing all constraints (solution candidates) implied by a given goal—followed by a simplification step—applying the solution selection criteria which identify the actual solution. This property enables translation of simplification-stratified programs into negation-stratified Datalog$^{\mathrm{LB}}$ programs defining the following sequence of operations:

1. based on the EDB facts and rules, derive the solution candidates (IDB facts)
2. identify the candidates that do not satisfy the program's solution selection criteria
3. determine the solution as the set of candidates not ruled out by the selection criteria.

We formally define the extended translation schema in Section 4.4. Here we illustrate the extension with an example CHR program and its Datalog$^{\mathrm{LB}}$ counterpart.

Example 4. Table 3(a) lists a simplification-stratified program that, given a set of numbers, finds its smallest element. The program rule iterates over the elements of the set, stored as individual constraints, and, upon finding one that is greater than some other set element, removes its representation from the constraint store. The Datalog$^{\mathrm{LB}}$ representation of the program is listed in Table 3(b). The program directly implements the three-step sequence outlined above. Line 1 identifies all set numbers as potential minimum elements, line 2 compares the numbers pairwise and adds all that are greater than some other number in the set to the predicate $\texttt{min}_{\neg}$, whereas line 3 identifies the actual set minimum as the element defined in `min` but not in $\texttt{min}_{\neg}$.

Many of the early formulations of CHR, e.g., [1,11], considered a simpagation rule of the form $\texttt{R@H}_1 \backslash \texttt{H}_2\texttt{<=>G|B}$ a syntactic abbreviation—and thus a semantic equivalent—of a simplification rule of the form $\texttt{R'@H}_1\texttt{,H}_2\texttt{<=>G|H}_1\texttt{,B}$. The notion of simplification stratification enables the following observation about this relationship. Given a

```
:- constraints min/1.

min(I) \ min(J) <=> J>=I | true.
```

(a)

```
min(I) <- min_ε(I).                                    1
min_¬(J) <- min(I), min(J), I<J.                       2
min_ω(I) <- min(I), !min_¬(I).                         3
```

(b)

Table 3. A CHR program finding the smallest number in a set (a), and its representation in Datalog$^{\mathrm{LB}}$ (b)

simplification-stratified program P comprising a rule R, by replacing the rule with its "equivalent" R′, we add to P's constraint dependency graph a self-loop (i.e., both incoming and outgoing) edge on each node representing a constraint in $\mathtt{H}_1$, and a negative incoming edge from each node representing a constraint in $\mathtt{H}_2$ to each node representing a constraint in $\mathtt{H}_1$. Clearly, the resulting program is not simplification stratified, meaning that, in general, the two kinds of rules are not equivalent.

4.2 Restricted function symbols

Datalog's termination guarantee follows from the fact that the interpretation of every predicate is a finite relation: for an n-ary predicate P, $P \subset U_1 \times U_2 \ldots \times U_n$ where each U_i is a finite Herbrand universe of 0-ary function symbols. Operationally, this enables evaluation of Datalog programs by a search through a finite number of interpretations. Extending Datalog with function symbols makes the Herbrand universe of terms infinite, and introduces the possibility of the logic engine exhaustively searching/enumerating an infinite space of solutions. Thus, pure Datalog—and our basic translation schema—disallow the use of function symbols of arity > 0. On the other hand, artithmetic functions, for instance, are very common in CHR, and hence relaxing this restriction may considerably expand the class of programs amenable for our translation.

Datalog$^{\mathrm{LB}}$ facilitates declaration of predicates as *functions* rather than just as relations, by specifying their domains and codomains: $p(d_1, \ldots, d_n, t_1, \ldots t_n)$. Such predicates are interpreted as functions: $d_1 \times \ldots \times d_n \mapsto t_1 \times \ldots \times t_n$. The Datalog$^{\mathrm{LB}}$'s type system verifies that the universes for all d_i are finite. With a finite domain, even if the interpretation of any t_i is infinite, the function itself is guaranteed to be finite as well.We exploit this feature of Datalog$^{\mathrm{LB}}$ to allow the use of infinite-domain functions in a way that preserves termination guarantees of pure Datalog.

Example 5. Table 4(a) lists a CHR program computing a distance from an arbitrary node in a tree to the tree's root (the depth of the node). The rule `root` sets the depth of the tree's root node to 0. The rule `node` repeatedly descends from a node to the node's child, and updates the depth counter. Even though the update applies an infinite-domain

```
:- constraints root/1, edge/2, depth/2.

root @ root(N) ==> depth(N,0).
node @ edge(N1,N2), depth(N1,D1) ==> D2 is D1+1, depth(N2,D2).
```

(a)

```
depth(N,0) <- root(N).
depth(N2,D2) <- edge(N1,N2), depth(N1,D1), D2 = D1+1.
```

(b)

Table 4. A CHR program calculating depth of a tree node (a), and its representation in Datalog$^{\mathrm{LB}}$ (b)

function ('+') to the counter value, the program is well-behaved. This is because the value of the counter functionally depends on the node, and so, the number of the increment operations is bound by the number of the nodes in the tree. The program translates directly to Datalog$^{\mathrm{LB}}$, and its representation is listed in Table 4(b).

4.3 Restricted unboundedness

The requirement that all pure-Datalog facts are ground complies with the original purpose of the language, which was to facilitate specification of database queries and (recursive) views, but makes pure Datalog inapplicable to problems that involve *top-down recursion* i.e., recursion through value-computing (in a broad sense) predicates. Since such problems are easily, and commonly, represented in CHR, searching for ways to relax this restriction seems worthwhile.

Example 6. Table 5(a) lists a CHR program encoding the naive union-find algorithm. The algorithm defines a forest of disjoint sets and two operations on its elements: *find* to determine which tree in the forest contains a given node, and *union* to merge two trees into one. A tree is represented by its root node. In the program, the constraints `root` and `->` capture the structure of the forest, whereas the constraints `make`, `union`, `find`, and `link` define the operations. The rule `make` creates a new tree with a single node, and designates that node as the tree's root. The rule `union`, given two nodes, merges the trees containing these nodes by finding the root of each tree and linking the two roots together. The rules `findNode` and `findRoot` repeatedly advance from a given node to its parent until reaching a root. The rules `linkEq` and `link` create a new tree by merging two existing root nodes, and designate one of these nodes as the tree's root.

The program in Table 5(a) defines recursive value computation by means of the `find` constraint which, given a node in a tree, returns the tree's root. The constraint is activated by the body of the `union` rule, with its first argument bound to the name of the node, and its second argument unbound. Activation of the constraint triggers either

```
:- constraints make/1, find/2, union/2, (->)/2, link/2, root/1.

make     @ make(A) <=> root(A).

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

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

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

(a)

```
:- constraints make/1, find/2, union/2, (->)/2, link/2, root/1,
               eq/2.

refl     @ eq(X,X) ==> true.
symm     @ eq(X,Y) ==> eq(Y,X).
trans    @ eq(X,Y), eq(Y,Z) ==> eq(X,Z).

make     @ make(A) <=> root(A).

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

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

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

(b)

Table 5. The naive union-find algorithm in traditional CHR encoding (a), and with user-defined unification (b)

the `findNode` or the `findRoot` rule, and only the latter unifies the second argument with the name of the tree's root node. Admitting unbound constraint arguments in a rule's body violates the range-restrictedness requirement of CHR$^{\delta}$, and, in the context of Datalog, leads to generation of non-ground facts, thus enforcing subgoal ordering. Hence, programs with top-down recursion cannot be represented in pure Datalog.

The translation of CHR programs with top-down recursion into Datalog$^{\mathrm{LB}}$ is enabled by a simple program transformation. The goal of the transformation is to replace the built-in constraint responsible for the delayed binding of the value argument with a user-defined constraint that can be added to the store (thus simulating the binding) at

any time of the evaluation. Transformed CHR programs are amenable for translation using the extended schema formalized in Section 4.4.

Example 7. Table 5(b) lists the transformed union-find program which defines a new constraint, `eq`, representing unification. The constraint is used in the body of the `findRoot` rule to reflect the binding of the variable to the name of the root node. Three new rules, `refl`, `symm` and `trans`, define the properties of the `eq` constraint.

4.4 Translation schema

The basic translation schema from Section 3.2 facilitates pure-Datalog representation of CHR programs required (i) to contain only range-restricted propagation rules free from function symbols, and (ii) to be evaluated only for ground queries. Clearly, these requirements demarcate a very small, and not very useful, subclass of CHR. In this section we define an extended translation schema for mapping a more interesting subclass of CHR into Datalog$^{\mathrm{LB}}$, by relaxing the basic schema's restrictions on program syntax in the following ways:

1. admit programs with simplification rules (governed by stratified simplification, Section 4.1)
2. admit rules with function symbols and local variables (governed by functional dependency, Section 4.2)
3. admit non-ground queries over programs (governed by user-defined built-ins, Section 4.3)

Hence, we generalize the notion of a rule amenable for translation:

Definition 9 (CHR$^{\chi}$ rule). *A* CHR$^{\chi}$ rule *is a CHR rule*

$$H_1 \setminus H_2 \iff G \mid A, B$$

where A *is a (possibly empty) sequence of arithmetic constraints;* B *is a sequence of user-defined and built-in (non-arithmetic) constraints; and for every constraint* $c \in A \cup B$*, each variable argument of* c *either appears in vars*$(H_1 \cup H_2)$*, or functionally depends on some variable in vars*$(H_1 \cup H_2)$*.*

Consequently, we generalize the definition of rule mapping:

Definition 10 (Rule mapping m_{χ}). *The mapping* $m_{\chi} : CHR^{\chi} \mapsto D$ *from CHR*$^{\chi}$ *rules to Datalog*$^{\mathrm{LB}}$ *clauses is defined as:*

$$m_{\chi}(H_1 \setminus H_2 \iff G \mid A, B) \quad = \quad B, H_{2\neg} \text{ :- } H_1, H_2, G, A$$

where the predicate $H_{2\neg}$ *denotes a sequence of the negated versions of all constraint symbols appearing in the head* H_2*:* $H_{2\neg} = \{c_{\neg} \mid c \in H_2\}$.

The predicate $H_{2\neg}$ simulates constraint removal. Since Datalog$^{\mathrm{LB}}$ does not support explicit removal of data, we denote removal of an instance of an IDB predicate `p` from a program's database by adding a corresponding fact to $p_{\neg}$. After completed evaluation, the database contains a set of facts added to `p`, in `p` itself, and a set of facts (marked as) removed from `p`, in $p_{\neg}$. We identify the facts that are actually in the database (i.e., are not marked as removed), by means of an *output predicate* and an *output rule*:

Definition 11 (Output predicate and output rule). *The* output predicate *for an IDB predicate p, p_ω, represents the actual definition of p in a program's database. The* output rule *for p, $r_\omega(p)$, defines the output predicate p_ω by selecting from the database the facts of p that have been added, but not removed, during the evaluation:*

$$r_\omega(p) \quad = \quad p_\omega \texttt{ <- } p\texttt{, !}p_\neg\texttt{.}$$

The set-semantics rules elimination, as well as the EDB predicates and EDB rules are preserved from the basic translation schema.

Definition 12 (Program translation). *A Datalog$^{\mathrm{LB}}$ translation of a CHR program given by a set of CHR$^\chi$ rules and a set of set-semantics rules over a set of user-defined constraints, $P(C) = R_\chi \cup R_\sigma$, is a program $m_\pi(P)$ in which each constraint $c \in C$ representing an IDB predicate is associated with an EDB rule and an output rule, each rule $r \in R_\chi$ is mapped to a Datalog$^{\mathrm{LB}}$ clause, and all set-semantics rules are eliminated: $m_\pi(P) = r_\epsilon(C) \cup r_\omega(C) \cup m_\chi(R_\chi) \cup m_\sigma(R_\sigma)$.*

Correctness The extended translation schema generates Datalog$^{\mathrm{LB}}$ programs which are operationally equivalent to the source CHR programs, and preserve well-behavedness properties of pure Datalog.

5 Full-fledged simplification

In Section 4.4 we defined a translation schema facilitating Datalog$^{\mathrm{LB}}$ representation of an interesting, but restricted, class of CHR programs. In this section we discuss one of the main challenges of translating into Datalog the unrestricted CHR.

The schema from Section 4.4 admits for translation into Datalog$^{\mathrm{LB}}$ CHR programs with stratified simplification, in which all constraint removals may be performed in a single (and final) evaluation step. In CHR programs that are not simplification stratified, simplification of constraints is interleaved with propagation, and it is possible that a constraint `c` added to the store at some point of the evaluation, is later removed, and then added again. Such behavior cannot be represented in Datalog$^{\mathrm{LB}}$ simply by means of a $\texttt{c}_\neg$ predicate as before, because this can capture at most one addition and removal of any given fact[2]. To reflect subsequent additions and removals, we need to extend the Datalog$^{\mathrm{LB}}$ predicate representing the constraint `c` with a time dimension, which will allow to keep track of the contents of the predicate at every point of program evaluation. We have implemented this approach by means of time stamps, or *steps*, added to each predicate in the translated program. With steps, a CHR simplification rule:

```
H <=> G | B
```

translates into a Datalog$^{\mathrm{LB}}$ rule:

```
B(SN), H(SN), H¬(SP) <- H(SP), G, next_step(SP,SN)
```

[2] recall that set semantics treats identical facts (constraint instances) as the same fact (constraint)

where `SP` and `SN` correspond to, respectively, the previous and the next step, and the predicate `next_step` advances from one step to the other. Simulating the time dimension in Datalog$^{\mathrm{LB}}$ takes away much of the programs' declarativeness, and leads to a significant increase in their state space. Furthermore, programs with time dimension are guaranteed to terminate only if the number of the steps taken is finite. To ensure this requires pre-allocating the steps at the beginning of the evaluation. If the number of steps necessary to fully process a problem cannot be determined statically, this may result in the evaluation terminating early, before reaching the result. Given these shortcomings, the benefit of pursuing this direction of our approach is yet to be determined.

6 Discussion and Related Work

In this paper, by establishing a relationship between CHR and two variants of Datalog, we have enabled reasoning about the logical reading of CHR programs in terms of the declarative and well-studied Datalog logic. Our basic translation schema shows a clean correspondence between pure Datalog and a small subclass of CHR. The programs in this CHR subclass are guaranteed to hold strong termination and confluence properties of pure Datalog. The extended translation schema significantly augments that basic subclass of CHR, at the same time preserving well-behavedness guarantees for the admitted programs. We expect to continue our investigation of this relationship, possibly enhancing it with the consideration of other relevant formalisms.

Connections between CHR and other formalisms have been studied before [1, 15–17]. First, a representation of CHR rules as universally quantified formulas in first-order predicate logic, together with a built-in constraint theory of the host language, defined a program's classical declarative semantics [1]. This approach turned out not always accurate (e.g., for programs with multiset semantics or procedural use of CHR[3]), opening the way to alternative interpretations. For instance, mapping CHR to intuitionistic linear logic [15] proved better suited for providing declarative semantics to programs with dynamic updates relying on non-deterministic committed choice. Even more accurate interpretation of procedural applications of CHR has been accomplished by transforming CHR programs into transaction logic [16], which uses a time-based dimension to reason at the level of individual derivation steps rather than only at the level of the final results. Our approach, relating CHR to (extended) Datalog, restores to the search for a declarative interpretation of CHR. Even though both pure Datalog and Datalog$^{\mathrm{LB}}$ lack the time-based dimension that allows to express full-fledged simplification, our translation enables declarative reasoning about a substantial subclass of CHR with restricted simplification, and we can simulate full-fledged simplification by extending the programs with a notion of pre-allocated evaluation steps.

CHR has been also related to (colored) Petri nets (CPN) [17]. The approach exploits a positive, range-restricted, ground subset of CHR, and proposes CPN-based analysis of concurrency properties of programs in this subset to facilitate parallelizing their execution. Our extended translation schema considers a larger subset of CHR (we relax both the range-restrictedness and groundness requirements), for which it guarantees confluence, thus opening the programs in this set to parallel processing.

[3] meaning, the use of CHR to express temporal, rather than purely logical, consequence

The connections between CHR and Datalog have been explored in the context of CHR$^\vee$ [18], an extension of CHR with disjunction, which facilitates CHR-based representation of (disjunctive) deductive databases. With support for mixing top-down and bottom-up programming paradigms, and admitting existentially quantified variables in rule bodies, the approach is an interesting complement to our work, and further studies of the relationship between the two seem worthwhile.

References

1. Frühwirth, T.: Theory and Practice of Constraint Handling Rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming **37**(1-3) (1998) 95–138
2. CHR. http://www.cs.kuleuven.ac.be/∼dtai/projects/CHR/
3. Lloyd, J.W.: Foundations of Logic Programming. Springer (1984)
4. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
5. Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: Symp. on Principles of Database Systems (PODS). (2005)
6. Loo, B., Condie, T., Garofalakis, M., Gay, D., Hellerstein, J., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking: Language, execution and optimization. In: Intl. Conf. on Management of Data (SIGMOD). (2006)
7. Li, N., Mitchell, J.: Datalog with constraints: A foundation for trust-management languages. In: Symp. on Practical Aspects of Declarative Languages (PADL). (2003)
8. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Logic **7**(3) (2006) 499–562
9. Ashley-Rollman, M.P., Rosa, M.D., Srinivasa, S.S., Pillai, P., Goldstein, S.C., Campbell, J.D.: Declarative programming for modular robots. In: Workshop on Self-Reconfigurable Robots/Systems and Applications. (2007)
10. White, W., Demers, A., Koch, C., Gehrke, J., Rajagopalan, R.: Scaling games to epic proportions. In: Intl. Conf. on Management of Data (SIGMOD). (2007)
11. Abdennadher, S.: Operational Semantics and Confluence of Constraint Propagation Rules. In: Intl. Conf. on Principles and Practice of Constraint Programming (CP). (1997)
12. Duck, G.J., Stuckey, P.J., de la Banda, M.J.G., Holzbaur, C.: The Refined Operational Semantics of Constraint Handling Rules. In: Intl. Conf. on Logic Programming (ICLP). (2004)
13. Sarna-Starosta, B., Schrijvers, T.: Indexing Techniques for CHR based on Program Transformation. Technical Report CW 500, Department of Computer Science, K.U.Leuven (2007)
14. Sarna-Starosta, B., Ramakrishnan, C.: Compiling Constraint Handling Rules for Efficient Tabled Evaluation. In: Symp. on Practical Aspects of Declarative Languages (PADL). (2007)
15. Betz, H., Frühwirth, T.: A linear-logic semantics for Constraint Handling Rules. In: Intl. Conf. on Principles and Practice of Constraint Programming (CP). (2005)
16. Meister, M., Djelloul, K., Robin, J.: A unified semantics for Constraint Handling Rules in transaction logic. In: Intl. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR). (2007)
17. Betz, H.: Relating coloured Petri nets to Constraint Handling Rules. In: Workshop on Constraint Handling Rules (CHR). (2007)
18. Abdennadher, S., Schütz, H.: CHR$^\vee$: A flexible query language. In: Intl. Conf. on Flexible Query Answering Systems (FQAS). (1998)