A transformational approach for proving properties of the CHR constraint store

Paolo Pilozzi*, Tom Schrijvers**, and Maurice Bruynooghe

Department of Computer Science, K.U.Leuven, Belgium
{Paolo.Pilozzi, Tom.Schrijvers, Maurice.Bruynooghe}@cs.kuleuven.be

Abstract. Proving termination of, or generating efficient control for Constraint Handling Rules (CHR) programs requires information about the kinds of constraints that can show up in the CHR constraint store. In contrast to Logic Programming (LP), there are not many tools available for deriving such information for CHR. Hence, instead of building analyses for CHR from scratch, we define a transformation from CHR to Prolog and reuse existing analysis tools for Prolog.
The proposed transformation has been implemented and combined with PolyTypes 1.3, a type analyser for Prolog, resulting in an accurate description of the types of CHR programs. Moreover, the transformation is not limited to type analysis. It can be used to prove any property of the constraints showing up in constraint stores, using tools for Prolog.
Keywords: Constraint Handling Rules, Program Transformation.

1 Introduction

Proving termination of, or generating efficient control for Constraint Handling Rules (CHR) programs requires information about the kinds of constraints that can show up in the CHR constraint store. In particular, type information is useful in this context. When used as a basis for determining the possible calls to the program, it leads to compiler optimisations [11], more precise termination conditions [4, 7] and more refined interpretations for proving termination [1, 9].

In Logic Programming (LP), many tools are available for performing such analyses [2, 5, 12]. Hence, instead of building analyses for CHR from scratch, it is interesting to explore whether one can define transformations from CHR to Prolog and reuse existing analysis tools for Prolog to obtain properties about the constraints that are in the CHR constraint store during computations.

One approach would be to build a faithful CHR meta-interpreter in Prolog and to analyse this meta-interpreter or to transform the CHR program into a Prolog meta-program and to analyse the meta-program. A difficulty with this approach is capturing the "fire-once" policy of CHR which prescribes that a rule cannot be applied twice to the same combination of constraints. This policy prevents the infinite application of propagation rules, that add constraints to the store without removing any. The approach in [10] has a problem with this.

* Supported by I.W.T. - Flanders (Belgium).
** Post-Doctoral Researcher of F.W.O. - Flanders (Belgium).

Fortunately, it often suffices to have an over-approximation of the constraints that can show up in the constraint store. In that case, one does not need a meta-interpreter or transformation that rigorously preserves the run-time behaviour of the CHR program and one can simply ignore the "fire-once" policy. This sometimes results in the presence of constraints in the approximated store that cannot be present at run-time, e.g., because some rule needs different occurrences of the same constraint before it can fire. But this is not too much of a problem, if only because one is typically interested in a whole class of queries (initial constraint stores), and queries in the class can have multiple occurrences of constraints, hence rules that need multiple occurrences can fire anyway.

For CHR, some direct approaches were developed [3, 11], mainly based on approaches developed for LP. Direct approaches usually make use of abstract interpretation. For CHR, not much work has been done on the topic of abstract interpretation [11] and thus not many analyses resulted from it. The transformational approach hasn't received much attention either. To the best of our knowledge, except for the termination preserving transformation discussed in [10], no transformational approaches have been attempted.

We have implemented the transformation and combined it with PolyTypes 1.3 [2] in a tool called CHRTypes. We plan to use CHRTypes as a source of information to obtain the call types of a CHR(Prolog) program. The computed call types can then be used as input to our termination analyser CHRisTA [8], instead of providing them ourselves, resulting in a fully automated termination analyser for CHR(Prolog).

The paper is organised as follows. In the next section we introduce CHR syntax and the abstract CHR semantics. Then, in Section 3, we discuss a transformation of CHR(Prolog) to Prolog. Section 4, discusses the application of our transformation to type analysis of CHR(Prolog), using PolyTypes 1.3 (based on [2]) on the transformed programs. Then, in Section 5, we evaluate our transformational approach using CHRTypes, a fully automated type analyser for CHR(Prolog). Finally, in Section 6, we conclude the paper.

2 Preliminaries

2.1 CHR Syntax

CHR is intended as a programming language for implementing constraint solvers. To implement these solvers, a user can define *CHR rules* which rewrite conjunctions of *constraints*. The constraints of a CHR program are special first-order predicates $c(t_1, \ldots, t_n)$ on terms, like the atoms of an LP program. There are two kinds of constraints defined in a CHR program: *CHR constraints* are user-defined and solved by the CHR program. *Built-in constraints* are pre-defined and solved by an underlying constraint theory, CT, defined in the host-language. We consider Prolog, thus definite LP with a left-to-right selection rule, as host-language. We assume the reader to be familiar with Prolog syntax and semantics.

A *CHR program*, P, is a finite set of *CHR rules*, defining the transitions of the program. To provide the analyser with information about the built-ins one can

add some Prolog clauses that capture their essential properties. In CHR, there are three different kinds of rules. *Simplification rules* replace CHR constraints by new CHR and built-in constraints. On the presence of CHR constraints, *propagation rules* only add new constraints. Finally, *simpagation rules* replace CHR constraints by new constraints, given the presence of other CHR constraints.

Let H_k, H_r and C denote conjunctions of CHR constraints and let G and B denote conjunctions of built-in constraints. Then, a simplification rule takes the form, $R \mathbin{@} H_r \Leftrightarrow G \mid B, C$, a propagation rule the form, $R \mathbin{@} H_k \Rightarrow G \mid B, C$, and a simpagation rule the form, $R \mathbin{@} H_k \setminus H_r \Leftrightarrow G \mid B, C$. Like in Prolog syntax, we write a conjunction of constraints as a sequence of conjuncts separated by commas. Rules are named by adding "*rulename* @" in front of the rule.

Example 1 (Merge-sort). The program below implements the merge-sort algorithm. The query $mergesort(L)$, with L a list of natural numbers of length exactly 2^n, yields a tree-representation of the order, which then is rewritten into a sorted list of elements. Note that in this version of merge-sort we represent the natural numbers using a symbolic form: $0, s(0), s(s(0)), \ldots$.

$$
\begin{array}{l}
R_1 \mathbin{@} msort([]) \Leftrightarrow true. \\
R_2 \mathbin{@} msort([L|Ls]) \Leftrightarrow r(0, L), msort(Ls). \\
R_3 \mathbin{@} r(D, L1), r(D, L2) \Leftrightarrow leq(L1, L2) \mid r(s(D), L1), a(L1, L2). \\
R_4 \mathbin{@} a(L1, L2) \setminus a(L1, L3) \Leftrightarrow leq(L2, L3) \mid a(L2, L3).
\end{array}
$$

The first two rules decompose a list of elements, while adding new $r/2$ constraints to the store. The constraints $r(D, L)$ represent trees of depth D (initially 0) and root value L. The third and fourth rule perform the actual merge-sorting. The third rule joins two trees of equal depth. It replaces both trees by a new tree of incremented depth, where the largest root becomes a child node of the smallest hence the branch is ordered. Note that the initial list needs to have a length that is a power of 2 to ensure that one ends with a single tree. The order in a branch is represented by $a/2$ constraints. Finally, the fourth rule merge-sorts different branches of a tree into a single branch, i.e., an ordered list of elements. □

2.2 The abstract CHR Semantics

In general, CHR is defined as a state transition system. In its simplest form, called the *abstract semantics*, it defines a state as a conjunction of constraints, called the *constraint store*. In it, there may be multiple identical constraints.

Definition 1 (CHR state). *A CHR state S is a conjunction of built-in and CHR constraints. An* initial state *or* query *is a finite conjunction of constraints. In a* final state *or* answer*, either the built-in constraints are inconsistent (*failed state*) or no more transitions are possible.* □

The rules of a CHR program determine the possible transitions between constraint stores. Since the abstract semantics ignores the fire-once policy, we have that all three kinds of rules are essentially simplification rules. Consider for

example the propagation rule, R @ $H_k \Rightarrow B, C$. Given the abstract CHR semantics, it is equivalent to the simplification rule, R @ $H_k \Leftrightarrow H_k, B, C$. Similarly, a simpagation rule, R @ $H_k \setminus H_r \Leftrightarrow B, C$, can be represented as a simplification rule, R @ $H_k, H_r \Leftrightarrow H_k, B, C$.

The transition relation relates consecutive CHR states on the presence of applicable CHR rules. The built-ins are non-deterministically solved by the CT.

Definition 2 (Transition relation). *Let θ denote a substitution corresponding to the bindings generated when resolving built-in constraints. Let σ denote a matching substitution of the variables in the head and an answer substitution of the variables appearing in the guard but not in the head. The* transition relation*, $\rightarrow$, between CHR states, given a constraint theory CT for the built-ins and a CHR program P for the CHR constraints, is defined as follows.*

1. ***Solve transition:***
   if $S = b \wedge S'$ *and* $CT \models b\theta$ ***then*** $S \rightarrow S'\theta$
2. ***Simplification:***
   given *a fresh variant of a rule in P:* $H_r \Leftrightarrow G \mid B, C$
   if $S = H'_r \wedge S'$ *and* $CT \models (H'_r = H_r\sigma) \wedge G\sigma$ ***then*** $S \rightarrow (B \wedge C \wedge S')\sigma$

We assume built-ins not to introduce new CHR constraints and thus solving these can only generate binding for variables. If built-in constraints cannot be solved by the CT, the CHR program fails. □

Note that by adding variable bindings to the constraint store (solving built-ins), a guard can become true. Also note that the selection of an answer substitution for the local variables in the guard is a committed choice. To denote the host-language CT we write CHR(CT), e.g. CHR(Prolog).

3 Transforming CHR(Prolog) to Prolog

In Section 2.2, we discussed the representation of the three kinds of CHR rules into simplification rules, thus safely over-approximating the contents of the constraint store with respect to the original theoretical CHR semantics. This choice was motivated in the introduction. We assume this transformation to take place prior to the transformation to Prolog that we discuss in this section.

3.1 Representing the CHR constraint store in Prolog

The CHR constraint store is a conjunction of constraints. To represent it in Prolog, we fix some order and represent it as a list, called the *storelist*. The code handling the firing of a rule will cope with the fact that the storelist is equivalent to any of its permutations. That there are $n!$ permutations for an n-element store is of no concern as the transformed program will be analysed, not executed.

Thus, for a constraint store $S = constr_1 \wedge constr_2 \wedge \ldots \wedge constr_n$, we obtain as a possible storelist representation $R = [constr_1, constr_2, \ldots, constr_n]$. Note that according to the abstract CHR semantics, a CHR query is an initial constraint store. Its representation by a storelist in Prolog is therefore identical to that of any other constraint store.

3.2 Representing CHR rules in Prolog

A CHR rule defines transitions between constraint stores. Which transitions are applicable for a constraint store, is determined by the presence of matching constraints for the heads of rules such that the guards of these rules are entailed. Multiple rules can be simultaneously applicable, in which case CHR commits to a particular choice. The following example illustrates this.

Example 2 (Non-determinism). Consider an initial store $a \wedge b$ for the program:

$$R_1 \ @ \ a \Leftrightarrow c. \qquad R_2 \ @ \ b \Leftrightarrow d. \qquad R_3 \ @ \ a, d \Leftrightarrow a, a, b.$$

The program may or may not end for the initial constraint store. If we apply the first rule, then the program terminates immediately. If we apply the second and third rule repeatedly, then the program runs forever. □

To model possible constraint stores that can exist during execution of a CHR program, it suffices to represent the non-determinism of CHR by search in Prolog. This is achieved by transforming every CHR rule to a Prolog clause of the $rule/2$ predicate. The clause describes the relationship between the store before and after rule application. To perform the matching between the store and the head of the rule, it is checked whether the storelist starts with the constraints in the rule head. Thus, a CHR rule of the form:

$$H_1, \ldots, H_n \Leftrightarrow G_1, \ldots, G_k \mid B_1, \ldots, B_l, C_1, \ldots, C_m$$

becomes a Prolog clause:

$$rule([H_1, \ldots, H_n|R], [B_1, \ldots, B_l, C_1, \ldots, C_m|R]) \ \texttt{:-} \ G_1, \ldots, G_k.$$

Here, $H_1, \ldots, H_n$ are head constraints. Built-in guards and bodies are represented respectively by $G_1, \ldots, G_k$ and $B_1, \ldots, B_l$. The CHR body constraints are represented by $C_1, \ldots, C_m$. Note that the head of the CHR rule is represented as a list with a variable as tail. This tail binds with the unused constraints in the current store. When the guards succeed, the new store consists of these unused constraints extended with the new constraints from the body.

As the CHR(Prolog) program has no rules for the built-in predicates, we need to add to the translation, rules that process them. For each built-in predicate p/n, there is therefore a clause $rule([p(X_1, \ldots, X_n)|R], R) \ \texttt{:-} \ p(X_1, \ldots, X_n)$ present in the transformed CHR(Prolog) program.

3.3 Representing the abstract semantics of CHR in Prolog

The operational semantics of CHR programs is already largely represented by the rule clauses. Matching of constraints in the store with heads of the rules is done by unification with the storelist. The resulting store is contained in the second argument of the rule clause. We only have to call rules repeatedly.

$$goal(S) \ \texttt{:-} \ perm(S, PS), rule(PS, NS), goal(NS). \qquad goal(_).$$

Note that we must permute the storelist – the call $perm(P, PS)$ – to bring the matching constraints to the front. Also note that whenever the program cannot call any of the $rule/2$ clauses, it will end up in a refutation, representing termination in CHR. In fact, any call to $goal/1$ can result in a refutation. Nevertheless, no further approximations of the contents of the CHR constraint store result from this. Finally, notice that CHR queries are represented by a call to $goal/1$ with a storelist representation of the CHR query as argument.

Example 3 (Transforming merge-sort). We revisit merge-sort from Example 1 and transform every rule into its clausal form. First, we represent all rules by simplification rules. This is already the case for the first three rules. The fourth rule on the other hand is a simpagation rule and is transformed into

$$R_4 \ @ \ a(L1, L2), a(L1, L3) \Leftrightarrow leq(L2, L3) \mid a(L1, L2), a(L2, L3).$$

Next, the CHR program is transformed into the following Prolog program.

$$\begin{array}{l}
goal(S) \ \texttt{:-} \ perm(S, PS), rule(PS, NS), goal(NS). \\
goal(_). \\
\\
rule([msort([])|R], R). \\
rule([msort([L|List])|R], [r(0, L), msort(List)|R]). \\
rule([r(D, L1), r(D, L2)|R], [r(s(D), L1), a(L1, L2)|R]) \ \texttt{:-} \ leq(L1, L2). \\
rule([a(L1, L2), a(L1, L3)|R], [a(L1, L2), a(L2, L3)|R] \ \texttt{:-} \ leq(L2, L3).
\end{array}$$

A query for the transformed program is of the form $goal([msort(L)])$, where L is a list of natural numbers in symbolic form, as in Example 1. □

3.4 Transformation Summary

To transform CHR states to Prolog queries, we introduce a mapping, $\alpha : S \rightarrow Q$, from a constraint store, S, to a Prolog query, Q, of the form $goal(R)$. Here, R is the storelist representation of S, as defined in Subsection 3.1. We define also the inverse of α as $\gamma = \alpha^{-1}$.

We introduce an operator, $C2P$, transforming a CHR program, P, to a Prolog program, $\wp$, and define it as follows.

Definition 3 ($C2P$). *A CHR program P is transformed into the following Prolog program $\wp = C2P(P)$.*

- *The Prolog program $\wp$ contains following clauses:*

$$\begin{array}{lll}
goal(S) \texttt{:-} & perm(L, [X|P]) \texttt{:-} & del(X, [Y|T], [Y|R]) \texttt{:-} \\
\quad perm(S, PS), & \quad del(X, L, L1), & \quad del(X, T, R). \\
\quad rule(PS, NS), & \quad perm(L1, P). & del(X, [X|T], T). \\
\quad goal(NS). & perm([], []). & \\
goal(_). & &
\end{array}$$

- *The Prolog program $\wp$ contains for every rule,*
  $$H_1, \ldots, H_n \Leftrightarrow G_1, \ldots, G_k \mid B_1, \ldots, B_l, C_1, \ldots, C_m.$$
  in P, where $B_1, \ldots, B_l$ are added built-in constraints and $C_1, \ldots, C_m$ are added CHR constraints, the following clause:
  $$rule([H_1, \ldots, H_n|R], [B_1, \ldots, B_l, C_1, \ldots, C_m|R]) \ :\text{-}\ G_1, \ldots, G_k.$$
- *The Prolog program $\wp$ contains for every built-in predicate p/n in P, a clause:*
  $$rule([p(X_1, \ldots, X_n)|R], R) \ :\text{-}\ p(X_1, \ldots, X_n).$$ □

We connect the CHR program, P, and its corresponding Prolog program, $\wp$, using α. We show that if a transition exists between two CHR states S and S', that there must exist a corresponding derivation in the transformed program. This derivation, however, is, in contrast to the CHR transition, no single-step operation. Between a call to $goal/1$ and a next call to $goal/1$, one needs to resolve the calls to $perm/2$ and $rule/2$, implementing the CHR transition. This property is illustrated in the diagram of Figure 1.

$$\begin{tikzcd} Q \arrow[r, "\wp"'] & Q' \arrow[d, "\gamma"] \\ S \arrow[u, "\alpha"] \arrow[r, "P"'] & S' \end{tikzcd}$$

Fig. 1. Connection $\wp$ and P. Here, Q and Q' are Prolog queries and S and S' CHR states. The vertical arrows represent mappings of α and γ. The horizontal arrows represent CHR transitions in P and derivations in $\wp$.

Theorem 1 (Connecting $\wp$ and P). *Consider CHR(Prolog). Let S be a constraint store and let Q be a goal statement such that $Q = \alpha(S)$. Let S' be the constraint store that is the result of applying a CHR transition on S, given by a CHR program P. Then, there exists a partial LD-derivation in $\wp = C2P(P)$ (SLD with Prolog's left to right selection rule) that starts in Q and leads to a goal Q' with the property that $S' = \gamma(Q')$.* □

Proof. We need to show that for every transition between consecutive states S and S' in the CHR program P, as given by the abstract CHR semantics, there exists a corresponding partial LD-derivation for the definite program $C2P(P)$ and the goal $Q = \alpha(S)$ that leads to some Q' with the property that $\gamma(Q') = S'$.

According to Definition 2, we can distinguish between two kind of transitions. We proof the property for each of them.

Solve transition In this case, the constraint store S holds a constraint b that refers to a built-in Prolog predicate b/n and that can be successfully solved by the underlying Prolog system. More precisely, the constraint store contains

$$\{b(X_1, \ldots, X_n)\sigma, c_1, \ldots, c_n\}$$

for some $n \geq 0$. Solving $b(X_1, \ldots, X_n)\sigma$ results in the application of a substitution θ and the resulting constraint store S' equals $\{c_1, \ldots, c_n\}\theta$.

The goal Q corresponding to the initial constraint store S is of the form

$$goal([a_1, \ldots, a_{n+1}])$$

with $[a_1, \ldots, a_{n+1}]$ a permutation of $[b(X_1, \ldots, X_n)\sigma, c_1, \ldots, c_n]$. When performing a resolution step on this goal with the program $C2P(P)$, either the fact $goal(_)$ is applied, yielding a refutation, or the clause

$$goal(A) \texttt{ :- } perm(A, Ap), rule(Ap, An), goal(An)$$

is applied, yielding the resolvent

$$perm([a_1, \ldots, a_{n+1}], Ap), rule(Ap, An), goal(An).$$

The latter is of interest to us. Resolving the subgoal $perm([a_1, \ldots, a_{n+1}], Ap)$ completely, leads to goals of the form

$$rule([b_1, \ldots, b_{n+1}], An), goal(An)$$

with $[b_1, \ldots, b_{n+1}]$ a permutation of $[a_1, \ldots, a_{n+1}]$ and hence a permutation of $[b(X_1, \ldots, X_n)\sigma, c_1, \ldots, c_n]$. Let us consider one with $b_1 = b$. In that case, we can write the goal as

$$rule([b(X_1, \ldots, X_n)\sigma, b_1 \ldots, b_n], An), goal(An)$$

with $[b_1 \ldots, b_n]$ a permutation of $[c_1, \ldots, c_n]$.
We can solve this goal with

$$rule([b(X_1, \ldots, X_n)|R], R) \texttt{ :- } b(X_1, \ldots, X_n),$$

yielding the new goal

$$b(X_1, \ldots, X_n)\sigma, goal([b_1 \ldots, b_n]).$$

Solving the built-in predicate results —by our assumptions about the constraint store— in the substitution θ and the new goal $goal([b_1 \ldots, b_n]\theta)$ with $[b_1 \ldots, b_n]\theta$ a permutation of $[c_1, \ldots, c_n]\theta$. Hence,

$$\gamma(goal([b_1 \ldots, b_n]\theta) = \{c_1, \ldots, c_n\}\theta = S'$$

which completes the proof for this case.

Simplification In this case, the constraint store S holds a number of constraints $h_1, \ldots, h_n$ that match the head of a CHR rule in P, i.e. it is of the form

$$\{h_1, \ldots, h_n, c_1, \ldots c_k\}$$

for some $k \geq 0$. The CHR rule is of the form

$$H_1, \ldots, H_n \Leftrightarrow G_1, \ldots, G_r \mid B_1, \ldots, B_l, C_1, \ldots, C_m.$$

Moreover, it is the case that there exists a matching substitution σ such that, for all $i \in [1..n]$ $H_i\sigma = h_i$ and that the guard $(G_1, \ldots, G_r)\sigma$ can be successfully solved, resulting in a substitution θ. As a next state, we thus obtain

$$S' = \{B_1, \ldots, B_l, C_1, \ldots, C_m\}\sigma\theta \uplus \{c_1, \ldots, c_k\}\theta.$$

The goal Q corresponding to the initial constraint store S is of the form

$$goal([a_1, \ldots, a_{n+k}])$$

with $[a_1, \ldots, a_{n+k}]$ a permutation of $[h_1, \ldots, h_n, c_1, \ldots, c_k]$. Similar as in the first case, we can apply the clause

$$goal(A) \texttt{ :- } perm(A, Ap), rule(Ap, An), goal(An)$$

and obtain the resolvent

$$perm([a_1, \ldots, a_{n+k}], Ap), rule(Ap, An), goal(An).$$

Completely resolving the subgoal $perm([a_1, \ldots, a_{n+k}], Ap)$ leads to goals of the form

$$rule([b_1, \ldots, b_{n+k}], An), goal(An)$$

with $[b_1, \ldots, b_{n+k}]$ a permutation of $[a_1, \ldots, a_{n+k}]$ and hence a permutation of $[h_1, \ldots, h_n, c_1, \ldots, c_k]$. Let us consider one with $b_1 = h1, \ldots, b_n = h_n$. In that case, we can write the goal as

$$rule([h_1, \ldots, h_n, b_{n+1} \ldots, b_{n+k}], An), goal(An)$$

with $[b_{n+1} \ldots, b_{n+k}]$ a permutation of $[c_1, \ldots, c_k]$.
We can solve this goal with

$$rule([H_1, \ldots, H_n|R], [B_1, \ldots, B_l, C_1, \ldots, C_m|R]) \texttt{ :- } G_1, \ldots, G_r;$$

we have that, for all i, $h_i = H_i\sigma$, hence we obtain the resolvent

$$G_1\sigma, \ldots, G_r\sigma, goal([B_1\sigma, \ldots, B_l\sigma, C_1\sigma, \ldots, C_m\sigma, b_{n+1} \ldots, b_{n+k}]).$$

By our assumptions about the store, resolving the guards $G_1\sigma, \ldots, G_n\sigma$ one by one results in an accumulated substitution θ and the new goal

$$goal([B_1\sigma, \ldots, B_l\sigma, C_1\sigma, \ldots, C_m\sigma, b_{n+1} \ldots, b_{n+k}]\theta)$$

with $[b_{n+1} \ldots, b_{n+k}]\theta$ a permutation of $[c_1, \ldots c_k]\theta$. Hence,

$$\gamma(goal([B_1\sigma, \ldots, B_l\sigma, C_1\sigma, \ldots, C_m\sigma, b_{n+1} \ldots, b_{n+k}]\theta) =$$
$$\{B_1\sigma, \ldots, B_l\sigma, C_1\sigma, \ldots, C_m\sigma, c_1, \ldots, c_k\}\theta = S'$$

which completes the proof for this case. □

This relation establishes that the analysis of properties of constraints, part of the CHR constraint store during execution of a CHR(Prolog) program, can take place on its transformed program $C2P(P)$ instead. After all, for every two consecutive CHR states, consecutive calls to $goal/1$ with storelist representations of these states exist. Stating the inverse is not true. For the transformed program, a refutation exists for every call to $goal/1$ and matching in CHR is replaced by the more general concept of unification in Prolog.

4 Application of the transformation to type analysis

In the previous section, we discussed the correctness of our transformation for the analysis of constraints present in constraint stores during computations of a CHR(Prolog) program P. That is, for every constraint store $\{c_1, \ldots, c_n\}$ that appears during the execution of P starting from some initial constraint store S, there is a corresponding goal $goal([c_1, \ldots, c_n])$ that appears in the execution of $C2P(P)$ starting from the goal $goal(\alpha(S))$. After introducing the notion of call set, we can formulate this in a more precise way as a corollary of the above theorem. Restricting the call set of $C2P(P)$ to the predicate $goal/1$ yields an over-approximation of the call set of the constraint store, defined similar to the call set of a Prolog program.

Definition 4 (Call set of a Prolog program). *Let S be a set of atomic goals. The call set, $Call(P, S)$, is the set of all atoms A, such that a variant of A is the selected atom in some derivation for (P, Q), for some $Q \in S$.* □

Corollary 1. *Let S be the initial constraint store of a CHR program P. Let C be the set $Call(C2P(P), \{\alpha(S)\})$ restricted to calls of the predicate $goal/1$. Then, $\{S' \mid \exists c \in C$ such that $\gamma(c) = S'\}$ is an over-approximation of the constraint stores that can occur during execution of the CHR program with S.* □

The corollary implies that the transformation preserves properties of the constraints in constraint stores that may occur during execution of the original CHR program. Although the transformation yields an over-approximation, it provides us with accurate information regarding the calls in the CHR program. Such information is derived top-down and does not depend much on the presence of a fire-once policy as argued in the introduction. After all, multiple applications of rules are considered anyways as we analyse the constraint store for classes of initial constraint stores. Approximations resulting from the refutations due to the fact $goal(_)$, do not influence the analysis of calls either.

Using unification instead of matching can introduce unwanted constraints. Consider for example the CHR rule, $a(1) \Leftrightarrow c$, and an initial constraint store $\{a(X)\}$, where X is some free variable. Then in CHR, the program would terminate immediately as the constraint in the store cannot be matched with the head of the rule. When transforming the CHR rule to a Prolog clause using $C2P$, we obtain: $rule([a(1)|R], [c|R])$; and for the the initial constraint store: $goal([a(X)])$. Due to unification in Prolog, the transformed program does allow the application of the transformed CHR rule. Consequently, a type analyser for Prolog will derive incorrectly that the constraint c can be part of the constraint store. Making the process of matching explicit would resolve the issue, however, doing so requires the use of Prolog built-ins. Type analysers for Prolog, typically, have a problem with this. In general, they do not take information of Prolog systems into account. Making matching explicit can thus only result in inaccurate types. This is in contrast to unification, where we might overestimate the constraints in constraint stores, but will compute the correct types.

A consequence of the corollary is that useful analyses of the resulting Prolog program are those that infer properties about the call set of the program. So, analyses that derive properties about the success set are useless (Unless combined with magic set transformation, so that the success set characterises the call set).

As an illustration of a useful analysis, below we apply the inference of well-typings [2] on the resulting Prolog programs. As well-typed programs cannot go wrong, all calls are well-typed and in particular, the type of the $goal/1$ predicate provides a well-typing for the constraints that can appear in the constraint store.

Other potentially useful analyses are mode analyses (combined with types). The modes of the $goal/1$ predicate provide information about the modes of the constraints in the store. Finally, a proof of termination of $goal(\alpha(S))$ is a sufficient condition for termination of the CHR program. However, as CHR programs with propagation are transformed into simplification only programs, we introduce non-termination. Therefore, proving termination on the transformed programs can only be done for programs without propagation [10].

The next example demonstrates a type analysis on C2P(merge-sort) from Example 1. First, all rules become simplification rules, as in Example 3. Then, the program is transformed according to Definition 3:

$$
\begin{array}{lll}
goal(S) :- & perm(L,[X|P]) :- & del(X,[Y|T],[Y|R]) :- \\
\quad perm(S,PS), & \quad del(X,L,L1), & \quad del(X,T,R). \\
\quad rule(PS,NS), & \quad perm(L1,P). & del(X,[X|T],T). \\
\quad goal(NS). & perm([],[]). & \\
goal(_). & &
\end{array}
$$

$$
leq(s(X),s(Y)) \ :- \ leq(X,Y). \qquad leq(0,X).
$$

$$
\begin{array}{l}
rule([msort([])|T],T). \\
rule([msort([L|List])|T],[r(0,L),msort(List)|T]). \\
rule([r(D,L1),r(D,L2)|T],[r(s(D),L1),a(L1,L2)|T]) \ :- \ leq(L1,L2). \\
rule([a(L1,L2),a(L1,L3)|T],[a(L1,L2),a(L2,L3)|T]) \ :- \ leq(L2,L3).
\end{array}
$$

Notice that we have added a definition for the built-in $leq/2$ for the sake of the analysis. Performing a type analysis on the transformed program with PolyTypes 1.3, yields the following result:

Type definitions:
$constraint \rightarrow msort(list); a(sym_nat_1, sym_nat_1); r(sym_nat_2, sym_nat_1)$
$list \rightarrow []; [sym_nat_1|list]$
$sym_nat_1 \rightarrow 0; s(sym_nat_1)$
$sym_nat_2 \rightarrow 0; s(sym_nat_2)$
$storelist \rightarrow []; [constraint|storelist]$

Signatures:
$goal(storelist)$
$perm(storelist, storelist)$
$del(constraint, storelist, storelist)$
$leq(sym_nat_1, sym_nat_1)$

For readability, we have replaced the type names generated by PolyTypes 1.3 by more meaningful ones. Note that sym_nat_1 and sym_nat_2 are equivalent types representing symbolic natural numbers. As these types do not interact through unification, the type inference keeps them separate.

For the analysis of types in CHR, we are only interested in the types present in the storelist of the transformed program. This is given by the signature for $goal/1$. It expresses that the type of its argument, is a *storelist*. That is, a list of elements of type *constraint*. Thus, terms of the form $msort(list)$, $a(sym_nat_1, sym_nat_1)$ or $r(sym_nat_2, sym_nat_1)$. Hence PolyTypes 1.3 correctly derives the types of the constraints that can occur in the storelist and thus in the constraint store.

The reason why PolyTypes is able to derive that we are using a list of symbolic integer values in $msort/1$ is because we have provided for the definition of $leq/2$. This is noticeable as its signature is present in the output generated by PolyTypes. Would we have not provided the implementation of $leq/2$, PolyTypes could not have derived the type definition $(sym_nat_1 \rightarrow 0; s(sym_nat_1))$ and would have concluded that $(sym_nat_1 \rightarrow)$ can take any type.

One could add a query to the program. Adding a query can only increase the type inferred by the PolyTypes analysis. For example, adding the CHR query $msort(l(s(s(s(0))), l(s(s(0)), n)))$, which translates into the Prolog query $goal([msort(l(s(s(s(0))), l(s(s(0)), n)))])$, will extend the type definition *list* as the argument of $msort/1$ in the query uses different list constructors than those in the *msort*-rules. That is, we use $l/2$ as list constructor yielding the type definitions $(list \rightarrow list_1; list_2)$, $(list_1 \rightarrow []; [sym_nat_1|list_1])$, and $(list_2 \rightarrow n; l(sym_nat_1, list_2))$. Actually, the obtained type is then a grave overestimation of the actual contents of the constraint store as no CHR rule can fire on the query. Here a call type analysis [6] would give more precise results.

Instead of specifying an initial query, one could also specify the type of the initial query, i.e. specifying that a call to $goal/1$ has the type *storelist* and providing for the initial type(s) for *constraint*, e.g. $constraint \rightarrow msort(list)$. Translating these types into input for PolyTypes, the tool will then extend these types and obtain the same types as the ones shown above.

5 Evaluation of the transformation

The transformation to Prolog from Definition 3 has been implemented and integrated with PolyTypes 1.3 in a tool called CHRTypes[1]. CHRTypes derives the types of CHR(Prolog) programs in a fully automated way, using only PolyTypes 1.3 and our transformation.

Using CHRTypes, we ran 10 tests on a benchmark of 98 CHR(Prolog) programs using a system with an $Intel(R)\ Pentium(R)\ D\ CPU\ 2.80GHz$ and $2G$ of RAM. In Table 1, we have listed the averages of these results for a representative subset of the CHR(Prolog) programs in the benchmark. Next to a set of

[1] Available at http://www.cs.kuleuven.be/∼paolo/c2p/

28 constructed examples, $compl_i$ and $constr_i$, the greater part of the benchmark consists of practical programs, among which the more complex CHR(Prolog) programs that we are aware of. The 37 example programs originating from WebCHR[2] consist of both small (such as *gcd*) and regular sized (such as *unionfind*) programs. The 33 designed by us consist of small (such as *revlist*), large (such as *genchrnet*), and large and complex (such as *gensccs*) programs.

CHR(swi)	**T(sec)**	**CHR(swi)**	**T(sec)**	**CHR(swi)**	**T(sec)**
ackermann	0.067	$color_3$	0.068	*concat*	0.066
$constr_{121}$	0.068	$constr_{122}$	0.062	*dcons2sat*	0.582
dfsearch	0.067	*dijkstra*	0.070	*fib_bu*	0.074
fib_td	0.068	*gcd*	0.068	*genchrnet*	0.178
gencss	0.181	*gensccs*	0.277	*hamming*	0.118
knapsack	0.101	*lazychr*	0.158	*linpoleq*	0.072
mergesort	0.082	*nqueen*	0.090	*oddeven*	0.073
power	0.067	$primes_3$	0.079	*revlist*	0.064
rsa	0.113	*shortest_path*	0.069	*solvecases*	0.203
strips	0.089	$trans_closure_1$	0.072	*unionfind*	0.087
uf_opt	0.152	*weight*	0.078	*ztoa*	0.065

Table 1. Benchmark results CHRTypes = C2P + PolyTypes 1.3

For all programs, CHRTypes computes the correct types within $0.6s$. For each program, a correct classification of signatures for CHR constraints and Prolog built-ins is given together with the correct set of type definitions.

In the next example, we show the output generated by CHRTypes for mergesort, however, implemented here for integers and not their symbolic counterparts. As such, we use =</2 instead of $leq/2$. The definition of =</2 is not made explicit in the program as it is provided by the host-language Prolog. Since PolyTypes does not make any assumptions on the argument types of built-in predicates, it cannot conclude on the types of the arguments of =</2.

Example 4 (Output of CHRTypes). CHRTypes generates the following output for integer merge-sort:

% Type definitions:
$(P1 \rightarrow) \quad (t42(P1) \rightarrow []; [P1|t42(P1)]) \quad (t37 \rightarrow 0; s(t37))$
% Signature CHR constraints:
$mergesort(t42(P1)) \quad edge(P1, P1) \quad root(t37, P1)$
% Signature Prolog Built-ins:
$true \quad P1 =< P1$
% Parsing: 0.0110s; Transforming: 0.0588s; PolyTypes 1.3: 0.0094s (Total: 0.0793s)

In the output of CHRTypes for this version of merge-sort (implemented for integers), the arguments of =</2, replacing the $leq/2$ predicate, can take any type. This is because =</2 is defined in the host-language, which PolyTypes does not take into account. Nevertheless, PolyTypes correctly infers that both arguments of =</2 must be of the same type. □

[2] http://chr.informatik.uni-ulm.de/∼webchr/

The benchmark results are very acceptable, however, could have been better given a different design of CHRTypes. Currently, CHRTypes integrates three separate tools. A parser computing more information than required. A transformer interpreting the result of the parser, while it could have been integrated with the parser. And finally, PolyTypes, which reads a Prolog file from disk to compute the types, requires the transformed program to be written to disk.

This shows in the results we obtained. Although we haven't included in Table 1 the timings for each separate tool, it is generally the case that the time required to compute the well-typing is not proportional to the time required to parse and to transform. Consider for example the following timings for $unionfind$ and gcd, both representative for a normal and toy program, respectively. The $unionfind$ program requires $0.012s$ to parse, $0.051s$ to transform, and $0.022s$ to compute the types. The gcd program requires much less time to compute the types, $0.009s$, but still needs a lot of time to parse, $0.011s$, and to transform, $0.045s$.

This can easily be overcome by better integration of the tools. The parser can easily be stripped of unnecessary components and could integrate with the transformation. PolyTypes could be adapted to accept a program as a list of clauses, avoiding as such the reading and writing of the Prolog program.

Although we demonstrated our transformation by a fully automated type analyser for CHR(Prolog), we could have easily adapted our system to other kinds of analyses of the calls in CHR programs, such as groundness information, modes and even call types. This would require only such a tool for Prolog programs and its integration into the current system.

6 Conclusion

We have presented a transformation from CHR(Prolog) programs to Prolog programs that respects the abstract CHR semantics. The transformed program describes transitions between storelists. Analysing it with respect to the storelist yields an overestimate of the CHR constraint store.

This way, existing tools for LP can be used to analyse the contents of the CHR constraint store. We have demonstrated this in the context of a type analysis, using the tool PolyTypes 1.3 integrated in our system CHRTypes, and obtained accurate type descriptions for the CHR constraints and Prolog built-ins of a CHR(Prolog) program. CHRTypes is therefore also applicable to pure Prolog programs providing the same accuracy as Polytypes on Prolog programs.

PolyTypes 1.3 does not take query information into account. There are however other tools which do so, such as the one in [5] for deriving call types. We have demonstrated that given a CHR query specification, there is a straightforward representation into a Prolog query specification for the transformed program. Essentially such a representation from CHR to Prolog corresponds to making the constraint store explicit as a list, enumerating the constraints in the store.

Our transformation does not prioritise on the rules to apply first. In most practical implementations, there is however some kind of a selection rule, e.g. based on rule orderings. In the context of termination this information is essential

to prove termination of certain programs. Future work will therefore be directed towards a better understanding of this problem. We will also apply our transformation to groundness and call type analysis and will, based on CHRTypes, develop a call type analyser for integration with CHRisTA [8], yielding the first fully automated termination analyser for the complete CHR(Prolog) language.

References

1. Maurice Bruynooghe, Michael Codish, John P. Gallagher, Samir Genaim, and Wim Vanhoof. Termination Analysis through Combination of Type Based Norms. *ACM Transactions on Programming Languages and Systems*, 29(2):10, 2007.
2. Maurice Bruynooghe, John P. Gallagher, and Wouter Van Humbeeck. Inference of Well-Typings for Logic Programs with Application to Termination Analysis. In *SAS '05: Proceedings of 12th International Statical Analysis Symposium*, pages 35–51, 2005.
3. Emmanuel Coquery and François Fages. A Type System for CHR. In *CHR '05: Proceedings of the 2nd International Workshop on Constraint Handling Rules*, pages 19–33, 2005.
4. Danny De Schreye and Stefaan Decorte. Termination of Logic Programs: The Never-Ending Story. *Journal of Logic Programming*, 19/20:199–260, 1994.
5. Gerda Janssens and Maurice Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2-3):199–260, 1992.
6. Manh Thang Nguyen. *Termination Analysis: Crossing Paradigm Borders*. PhD thesis, Katholieke Universiteit Leuven - Departement Computer Wetenschappen, Belgium, 2009.
7. Paolo Pilozzi and Danny De Schreye. Termination analysis of CHR revisited. In *ICLP '08: Proceedings of the 24th International Conference on Logic Programming*, pages 501–515, 2008.
8. Paolo Pilozzi and Danny De Schreye. Automating termination proofs for CHR. In *ICLP '09: Proceedings of the 25th International Conference on Logic Programming*, pages 504–508, 2009.
9. Paolo Pilozzi and Danny De Schreye. Proving termination by invariance relations. In *ICLP '09: Proceedings of the 25th International Conference on Logic Programming*, pages 499–503, 2009.
10. Paolo Pilozzi, Tom Schrijvers, and Danny De Schreye. Proving termination of CHR in Prolog: A transformational approach. In *WST '07: Proceedings of the 9th International Workshop on Termination*, 2007.
11. Tom Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules*. PhD thesis, Katholieke Universiteit Leuven, Departement Computer Wetenschappen, Belgium, 2005.
12. Tom Schrijvers, Maurice Bruynooghe, and John P. Gallagher. From Monomorphic to Polymorphic Well-Typings and Beyond. In *LOPSTR '08: Pre-proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation*, pages 152–167, 2008.