Abstract Interpretation for Constraint Handling Rules

Tom Schrijvers[*]
Department of Computer Science
Katholieke Universiteit Leuven, Belgium
toms@cs.kuleuven.ac.be

Peter J. Stuckey
NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
pjs@cs.mu.oz.au

Gregory J. Duck
Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
gjd@cs.mu.oz.au

ABSTRACT

Program analysis is essential for the optimized compilation of Constraint Handling Rules (CHRs) as well as the inference of behavioral properties such as confluence and termination. Up to now all program analyses for CHRs have been developed in an ad hoc fashion.

In this work we bring the general program analysis methodology of abstract interpretation to CHRs: we formulate an abstract interpretation framework over the call-based operational semantics of CHRs. The abstract interpretation framework is non-obvious since it needs to handle the highly non-deterministic execution of CHRs. The use of the framework is illustrated with two instantiations: the CHR-specific *late storage* analysis and the more generally known groundness analysis. In addition, we discuss optimizations based on these analyses and present experimental results.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Constraint and logic languages*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms, Languages, Performance

Keywords

Abstract interpretation, Constraint Handling Rules

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

1. INTRODUCTION

Constraint Handling Rules (CHRs) [4] are a rule-based language developed for expressing constraint solvers.

Although the language has existed for several years now and has a reasonable reference implementation in SICStus Prolog [7], there have been relatively few implementations. The recent appearance of new CHR systems [2, 11] has given rise to the need to communicate and compare between different CHR systems has given rise to the formulation of the more deterministic refined operational semantics [3] shared among CHR compilers.

Apart from the common formal semantics to be implemented by CHR compilers, 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.

Abstract interpretation [1] is a general methodology for program analysis by abstractly executing the program code. Abstract interpretation provides a remedy for the current difficulties in correctly analyzing CHR programs, and should enable optimizing CHR compilers to reach a new level of complexity and correctness.

Overview First, in Section 2, we briefly introduce CHRs and its relevant concepts. Second, Section 3 presents the call-based refined operational semantics of CHRs that will be abstractly interpreted. The general abstract interpretation framework is then defined in Section 4. Two instances of the framework, late storage analysis and groundness analysis, illustrate the framework in Sections 5 and 6 respectively. The implementation and experimental evaluation of these analyses are subsequently reported on in Section 7. Finally, we conclude in Section 8.

2. CONSTRAINT HANDLING RULES INTRODUCTION

In this section we briefly introduce CHRs. For a more thorough overview of CHRs we refer the reader to [4].

2.1 CHRs by Example

The set of constraint handling rules below defines a less-than-or-equal constraint (`leq/2`) over numbers. The rules illustrate several syntactical features of CHRs.

```
leq(X,X) <=> true.
leq(X,Y) <=> number(X), number(Y) | X =< Y.
leq(X,Y), leq(Y,X) <=> X = Y.
leq(X,Y) \ leq(X,Y) <=> true.
leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

The first, second and third rule are simplification rules, indicated by the double arrow `<=>`. To the left of the arrow is the *head* of the rule, while to the right is the *body*. A simplification rule has the meaning that constraints matching the head can be replaced by those in the body. The meaning of this first rule should be clear: the `leq` relation is reflexive, and hence `leq(X,X)` is trivially satisfied and bears no information, so it can be removed (`true` represents the empty set of constraints).

The second rule shows that a rule can be extended by a guard, after the arrow (`<=>`) and before the vertical bar `|`. In this case the guard is `number(X), number(Y)`. The replacement defined by the rule only occurs for constraints that match the head and satisfy the guard. The guards are technically constraints that can be checked for entailment by the underlying constraint system. In practice for CHRs defined over logic programming languages they are goals that do not constrain variables of the head. Rule two replaces the constraint `leq(X,Y)` with a simple built-in inequality check `X =< Y` if the arguments are bound to numbers.

The third rule illustrates that the head of a rule can contain a conjunction of multiple constraints. It formulates the antisymmetry property of the `leq` constraint.

The fifth rule with the `==>` is a propagation rule. The rule states that if we find constraints matching the head we should add the constraints in the body. We should only do this once for each combination.

The fourth rule is a "simpagation" rule. It has the same meaning as a simplification rule where the constraints before the backslash would be called again in the body. However it is more efficient in that it never removes those head constraints and does not unnecessarily trigger more rules in that way. In the `leq` constraint definition its role is to declare the set semantics of the constraint, i.e. the number of copies of a constraint is not important and hence it is more efficient to keep only one.

2.2 CHR Semantics

The initial, theoretical operational semantics of CHRs [4] was essentially a multiset rewriting system. In [3] this semantics is defined as a transition system ω_t.

The semantics ω_t is highly non-deterministic, hence their high level nature. The non-determinism is caused by several factors. Firstly, several transition rules may be applicable at any particular state; any of them may be chosen. Secondly, the transition rules themselves are non-deterministic. At any stage many different matches for a CHR rule may apply, and the choice of which built-in or CHR constraint to add to the store at any stage is open.

Although this ω_t semantics is highly non-deterministic, actual CHR compilers typically already resolve most of this non-determinism statically. In fact, the refined operational semantics ω_r [3] reflects a large part of the increased determinism that is present in most CHR compilers we are aware of.

The ω_r semantics is more involved, yet it no longer leaves any non-determinism in what transition rule should be applied in what state. Furthermore, it limits the constraint sequences to which a particular rule can be applied in a particular state. The order in which the execution stack is processed, is also fixed. The only remaining non-determinism is in the order in which triggered constraints are added to the execution stack by the the addition of a built-in constraint to the store, and the order in which matching partner constraints are tried in a rule.

3. THE CALL-BASED REFINED OPERATIONAL SEMANTICS ω_c

In this section we present the call-based refined operational semantics ω_c. It is a variant of the refined operational semantics ω_r [3] designed to make the analysis more straightforward. For the analysis of logic programs, we do not directly analyse over the derivations based operational semantics, instead we introduce a call based semantics which makes the number of abstract goals to be considered finite (see e.g. [8]). We introduce the call-based refined operational semantics for CHRs for the same reason. We show in [5] that ω_c and ω_r are equivalent.

The main difference between the two semantics lies in their formulation. The transition system of ω_r linearizes the dynamic call-graph of CHR constraints into the execution stack of its execution states. In ω_c on the other hand constraints are treated as procedure calls: each newly added *active* constraint searches for possible matching rules in order, until all matching rules have been executed or the constraint is deleted from the store. As with a procedure, when a matching rule fires other CHR constraints may be executed as subcomputations and, when they finish, the execution returns to finding rules for the current active constraint. The latter semantics is much closer to the procedure-based target languages, like Prolog and HAL, of the current CHR compilers.

We believe that this makes the ω_c semantics much more suitable for reasoning about optimizations. After all, optimizations are typically formulated on the level of the generated code in the target language.

A CHR *program* P is a sequence of CHR rules. The (call-based) refined semantics for CHRs uses the notion of an active constraint to determine which rules will be tested for firing. The active constraint is checked against each of the occurrences for its predicate in the program in turn. This leads us to the keep track of occurrences of predicates. We assume that each head constraint is numbered from 1, in a top-down right-to-left manner. The numbered version of the `leq` program is:

```
leq(X,X)₁ <=> true.
leq(X,Y)₂ <=> number(X), number(Y) | X =< Y.
leq(X,Y)₄, leq(Y,X)₃ <=> X = Y.
leq(X,Y)₆ \ leq(X,Y)₅ <=> true.
leq(X,Y)₈, leq(Y,Z)₇ ==> leq(X,Z).
```

The rest of this section is structured as follows. In Sections 3.1 and 3.2 we respectively present the execution state and transition rules of ω_c. Section 3.3 illustrates the semantics on an example.

3.1 Execution State of ω_c

Formally, the execution state of the call-based refined semantics is the tuple $\langle G, A, S, B, T\rangle_n$ where G, A, S, B, T and n, representing the goal, call stack, CHR store, built-in store, propagation history and next free identity number respectively. We define the domain of execution states to be Σ and will denote elements as $\sigma, \sigma_0, \sigma_1, \ldots$.

The *goal* G is a sequence of CHR constraints and built-in constraints. We use $\Box$ to denote the empty sequence, and write it as `true` in programs.

An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with some unique integer i. This number serves to differentiate among copies of the same constraint. We introduce functions $chr(c\#i) = c$ and $id(c\#i) = i$, and extend them to sequences and sets of identified CHR constraints in the obvious manner.

An *occurrenced* identified CHR constraint $c\#i : j$ indicates the identified CHR constraint is being considered for matches at occurrence j of constraint c.

The *execution stack* A is a sequence of constraints, identified CHR constraints and occurrenced identified CHR constraints.

The *CHR store* S is a set of identified CHR constraints.

The *built-in constraint store* B contains any built-in constraint that has been passed to the underlying solver. We assume $\mathcal{D}$ is the constraint theory for the underlying solver.

The *propagation history* T is a set of sequences, each recording the identities of the CHR constraints which fired a rule, and the name of the rule itself. 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 to fire the rule before. Finally, the *next free identity* n represents the next integer which can be used to number a CHR constraint.

Given initial goal G, the initial state is $\langle G, \Box, \emptyset, \emptyset, \emptyset\rangle_1$.

The function *pp* returns the *program point* of an execution state:

$$\begin{array}{rcll}
pp(\langle G, A, S, B, T\rangle_n) & = & pp(G) & \\
pp(c) & = & \texttt{builtin} & \text{(c built-in)} \\
pp(p(x_1, \ldots, x_n)) & = & p/n & \\
pp(p(x_1, \ldots, x_n)\#i) & = & p/n & \\
pp(p(x_1, \ldots, x_n)\#i : j) & = & p/n : j & \\
pp([c_1, \ldots, c_n]) & = & [pp(c_1), \ldots, pp(c_n)] &
\end{array}$$

The program point relates the execution state to compiled code (see e.g. [6]): the program point p/n corresponds with the code for the **Activate** transition of constraint p/n and the program point $p/n : i$ corresponds with the code for occurrence i.

3.2 Transition Rules of ω_c

Execution proceeds by exhaustively applying transitions to the initial execution state until the built-in solver state is unsatisfiable or no transitions are applicable.

We define transitions from state σ_0 to σ_1 as $\sigma_0 \rightarrowtail_N \sigma_1$ where N is the (shorthand) name of the transition. We let $\rightarrowtail^*$ be the reflexive transitive closure of $\rightarrowtail$ (for all names N). We let $\uplus$ denote multiset union and $++$ denote sequence concatenation. Let $vars(o)$ be the variables in object o. We use $\exists_{\{v_1,\ldots,v_n\}}$ to mean $\exists v_1 \cdots \exists v_n$. We use $\bar{\exists}_V F$ to mean $\exists_{vars(F)-V}$, that is quantifing away all variables not in V. The possible transitions are as follows:

1. Solve $\langle c, A, S, B, T\rangle_n \rightarrowtail_{So} \langle \Box, A, S', B', T'\rangle_{n'}$ where c is a built-in constraint. If $\mathcal{D} \models \neg\bar{\exists}_\emptyset(c \wedge B)$, then $S' = S$, $B' = c \wedge B$, $T' = T$, $n' = n$. Otherwise ($\mathcal{D} \models \bar{\exists}_\emptyset(c \wedge B)$), where

$$\langle S_1, A, S, c \wedge B, T\rangle_n \rightarrowtail^* \langle \Box, A, S', B', T'\rangle_{n'}$$

and $S_1 = solve(S, B, c)$ is a subset of S satisfying the following conditions:

1. *lower bound*: For all $M = H_1 ++ H_2 \subseteq S$ such that there exists a rule $r \in P$:

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

   and a substitution θ such that

   $$\begin{cases} chr(H_1) = \theta(H_1') \\ chr(H_2) = \theta(H_2') \\ \mathcal{D} \models \neg(B \rightarrow \exists_{vars(r)}(\theta \wedge g)) \wedge (B \wedge c \rightarrow \exists_{vars(r)}(\theta \wedge g)) \end{cases}$$

   then $M \cap S_1 \neq \emptyset$

2. *upper bound*: If $m \in S_1$ then $vars(m) \not\subseteq fixed(B)$, where $fixed(B)$ is the set of variables fixed by B.

The actual definition of the *solve* function will depend on the underlying solver.

2a. Activate $\langle c, A, S, B, T\rangle_n \rightarrowtail_A \langle c\#n : 1, A, \{c\#n\} \uplus S, B, T\rangle_{(n+1)}$ where c is a CHR constraint which has never been active.

2b. Reactivate $\langle c\#i, A, S, B, T\rangle_n \rightarrowtail_R \langle c\#i : 1, A, S, B, T\rangle_n$ where $c\#i$ is a CHR constraint in the store (back in the queue through **Solve**).

3. Drop $\langle c\#i : j, A, S, B, T\rangle_n \rightarrowtail_{Dp} \langle \Box, A, S, B, T\rangle_n$ where $c\#i : j$ is an occurrenced active constraint and there is no such occurrence j in P.

4. Simplify

$$\langle c\#i : j, A, \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail_{Si} \langle \Box, A, S', B', T''\rangle_{n'}$$

where

$$\langle \theta(C), A, H_1 \uplus S, \theta \wedge B, T'\rangle_n \rightarrowtail^* \langle \Box, A, S', B', T''\rangle_{n'}$$

where the j^{th} occurrence of the CHR predicate of c is d_j in rule $r \in P$:

$$H_1' \setminus H_2', d_j, H_3' \iff g \mid C$$

and there exists matching substitution θ is such that $c = \theta(d_j)$, $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$, $chr(H_3) = \theta(H_3')$, and $\mathcal{D} \models B \rightarrow \bar{\exists}_{vars(r)}(\theta \wedge g)$, and the tuple $id(H_1) ++ [i] ++ id(H_2) ++ id(H_3) ++ [r] \notin T$. The substitution θ must also rename apart all variables appearing only in g and C. In the intermediate transition sequence $T' = T \cup \{id(H_1) ++ id(H_2) ++ [i] ++ id(H_3) ++ [r]\}$.

If no such matching substitution exists then

$$\langle c\#i : j, A, S, B, T\rangle_n \rightarrowtail_{Si} \langle c\#i : j+1, A, S, B, T\rangle_n$$

5. Propagate

$$\langle c\#i : j, A, \{c\#i\} \uplus S, B, T\rangle_n \rightarrowtail_P \langle G, A, S_k, B_k, T_k\rangle_{n_k}$$

where the j^{th} occurrence of the CHR predicate of c is d_j in rule $r \in P$:

$$H_1', d_j, H_2' \setminus H_3' \iff g \mid C$$

Let $S_0 = S \uplus \{c\#i\}, B_0 = B, T_0 = T, n_0 = n$.

Now assume, for $1 \leq l \leq k$ and $k \geq 0$, the series of transitions

$$\langle C_l, [c\#i : j|A], H_{1l} \uplus \{c\#i\} \uplus H_{2l} \uplus R_l, B_{l-1}, T_{l-1} \cup \{t_l\}\rangle_{n_{l-1}} \\ \rightarrowtail^* \langle \Box, [c\#i : j|A], S_l, B_l, T_l\rangle_{n_l}$$

where $\{c\#i\} \uplus H_{1l} \uplus H_{2l} \uplus H_{3l} \uplus R_l = S_{l-1}$ and there exists a matching substitution θ_l such that

$$\begin{cases} c = \theta_l(d_j) \\ C_l = \theta_l(C) \\ chr(H_{1l}) = \theta_l(H'_1) \\ chr(H_{2l}) = \theta_l(H'_2) \\ chr(H_{3l}) = \theta_l(H'_3) \\ \mathcal{D} \models B_{l-1} \rightarrow \bar{\exists}_{\theta_l(r)}\theta_l(g) \\ t_l = id(H_{1l}) +\!\!+ [i] +\!\!+ id(H_{2l}) +\!\!+ id(H_{3l}) +\!\!+ [r] \not\in T_{l-1} \end{cases}$$

where θ_l renames apart all variables only appearing in g and C (separately for each l).

Furthermore, for $k+1$ no such transition is possible.

The resulting goal G is either $G = \Box$ if $\mathcal{D} \models \neg\bar{\exists}_\emptyset B_k$ (i.e. failure occurred) or $G = c\#i : j+1$ otherwise.

The role of the propagation histories T_l is exactly the same as with the theoretical operational semantics, ω_t, to prevent the same propagation rule firing twice.

6. Goal

$$\langle [c|C], A, S, B, T\rangle_n \rightarrowtail_G \langle G, A, S', B', T'\rangle_{n'}$$

where $[c|C]$ is a sequence of built-in and CHR constraints

$$\langle c, A, S, B, T\rangle_n \rightarrowtail^* \langle \Box, A, S', B', T'\rangle_{n'}$$

and $G = \Box$ if $\mathcal{D} \models \bar{\exists}_\emptyset(\neg B')$ (i.e. calling c caused failure) or $G = C$ otherwise. $\Box$

3.3 Example

Now we illustrate the call-based semantics on a small example program:

```
p₁ ==> q.
p₂, t₁ <=> r.
p₃, r₁ ==> true.
p₄ ==> s.
p₅, s₁ <=> true.
```

All the occurrences of constraints in the above program are indexed with their respective occurrence numbers. Starting from an initial goal p the derivation under the call-based refined operational semantics goes as follows (for brevity we omit the propagation history, denoted by $\bullet$):

For both the simplification Si and propagation rules P we annotate the name with $\neg$ if the rule did not find a match.

$$\begin{array}{ll}
 & \langle \mathtt{p}, [], \emptyset, \emptyset, \bullet\rangle_1 \\
\rightarrowtail_A & \langle \mathtt{p\#1:1}, [], \{\mathtt{p\#1}\}, \emptyset, \bullet\rangle_2 \\
\rightarrowtail_P & \langle \mathtt{p\#1:2}, [], \{\mathtt{p\#1}, \mathtt{q\#2}\}, \emptyset, \bullet\rangle_2 \\
 & \qquad \langle \mathtt{q}, [\mathtt{p\#1:1}], \{\mathtt{p\#1}\}, \emptyset, \bullet\rangle_2 \\
 & \qquad \rightarrowtail^* \langle \Box, [\mathtt{p\#1:1}], \{\mathtt{p\#1}, \mathtt{q\#2}\}, \emptyset, \bullet\rangle_3 \\
\rightarrowtail_{\neg Si} & \langle \mathtt{p\#1:3}, [], \{\mathtt{p\#1}, \mathtt{q\#2}\}, \emptyset, \bullet\rangle_3 \\
\rightarrowtail_{\neg P} & \langle \mathtt{p\#1:4}, [], \{\mathtt{p\#1}, \mathtt{q\#2}\}, \emptyset, \bullet\rangle_3 \\
\rightarrowtail_P & \langle \mathtt{p\#1:5}, [], \{\mathtt{q\#2}\}, \emptyset, \bullet\rangle_4 \\
 & \qquad \langle \mathtt{s}, [\mathtt{p\#1:4}], \{\mathtt{p\#1}, \mathtt{q\#2}\}, \emptyset, \bullet\rangle_3 \\
 & \qquad \rightarrowtail^* \langle \Box, [\mathtt{p\#1:4}], \{\mathtt{q\#2}\}, \emptyset, \bullet\rangle_4 \\
\rightarrowtail_{\neg Si} & \langle \mathtt{p\#1:6}, [], \{\mathtt{q\#2}\}, \emptyset, \bullet\rangle_4 \\
\rightarrowtail_{Dp} & \langle \Box, [], \{\mathtt{q\#2}\}, \emptyset, \bullet\rangle_4
\end{array}$$

The full subderivation executing q in the body of the first rule is:

$$\begin{array}{ll}
 & \langle \mathtt{q}, [\mathtt{p\#1:1}], \{\mathtt{p\#1}\}, \emptyset, \bullet\rangle_2 \\
\rightarrowtail_A & \langle \mathtt{q\#2:1}, [\mathtt{p\#1:1}], \{\mathtt{p\#1}, \mathtt{q\#2}\}, \emptyset, \bullet\rangle_3 \\
\rightarrowtail_{Dp} & \langle \Box, [\mathtt{p\#1:1}], \{\mathtt{p\#1}, \mathtt{q\#2}\}, \emptyset, \bullet\rangle_3
\end{array}$$

And the full subderivation executing s in the body of the fourth rule is:

$$\begin{array}{ll}
 & \langle \mathtt{s}, [\mathtt{p\#1:4}], \{\mathtt{p\#1}, \mathtt{q\#2}\}, \emptyset, \bullet\rangle_3 \\
\rightarrowtail_A & \langle \mathtt{s\#3:1}, [\mathtt{p\#1:4}], \{\mathtt{p\#1}, \mathtt{q\#2}, \mathtt{s\#3}\}, \emptyset, \bullet\rangle_4 \\
\rightarrowtail_{Si} & \langle \Box, [\mathtt{p\#1:4}], \{\mathtt{q\#2}\}, \emptyset, \bullet\rangle_4 \\
 & \qquad \langle \Box, [\mathtt{s\#3:1}, \mathtt{p\#1:4}], \{\mathtt{p\#1}, \mathtt{q\#2}, \mathtt{s\#3}\}, \emptyset, \bullet\rangle_4 \\
 & \qquad \rightarrowtail^* \langle \Box, [\mathtt{s\#3:1}, \mathtt{p\#1:4}], \{\mathtt{p\#1}, \mathtt{q\#2}, \mathtt{s\#3}\}, \emptyset, \bullet\rangle_4
\end{array}$$

4. ABSTRACT INTERPRETATION FRAMEWORK

In this section we present our generic abstract interpretation framework for CHRs. The framework for CHRs is based on an abstraction of the operational semantics given in the previous section. Instead of a concrete state, an abstract state is used and similarly, abstract transition rules are used instead of concrete ones.

In Sections 4.1 and 4.2 we discuss how a particular instance of the framework, i.e. an analysis domain, should specify its abstract state and abstract transition rules.

The generic, domain-independent aspects of the abstract semantics, which are provided by the framework, are presented in Section 4.3. It covers how the framework applies the abstract transition rules starting from what initial state and how the framework deals with non-determinism.

4.1 Abstract State

Every instance of the abstract interpretation framework should define a domain Σ_a of abstract states. The abstract domain Σ_a has to be a lattice with partial ordering $\preceq$, least upper bound $\sqcup$ and greatest lower bound $\sqcap$ operations.

Furthermore an abstraction function α has to be defined from a concrete state σ (as defined in Section 3.1) to an abstract state s, and a concretisation function γ from an abstract state to a set of concrete states.[1]

We impose an additional restriction on γ:

$$\forall s \in \Sigma_a : \forall \sigma_1, \sigma_2 \in \gamma(s) : pp(\sigma_1) = pp(\sigma_2)$$

i.e. every abstract execution state should correspond with exactly one program point. This allows us to extend the domain of the pp function to abstract states:

$$pp(s) = pp(\sigma) \text{ with } \sigma \in \gamma(s)$$

The restriction is imposed for two reasons:

- to be able to associate analysis information contained in abstract states with the program points, and
- to determine whether a particular abstract state s is a final state (i.e. $pp(s) = \Box$).

The accurate program point information may complicate the abstract semantics somewhat, but results in more accurate analyses.

The framework will only make use of the least upper bound operation $s_1 \sqcup s_2$ on states corresponding to the same program point ($pp(s_1) = pp(s_2)$).

4.2 Abstract Transition Rules

The abstract domain must provide the following abstract operations corresponding to the transitions in the call based

[1]Typically we only specify α and assume γ to be defined as $\gamma(s) = \{\sigma | \alpha(\sigma) = s\}$.

semantics: AbstractSolve, AbstractActivate, AbstractReactivate, AbstractDrop, AbstractSimplify, AbstractPropagate and AbstractGoal. These abstract operations are abstractions of the transition rules defined by the call-based refined operational semantics of CHRs, as given in Section 3.2.

With the exception of AbstractSimplify, the abstract transitions are transitions of the form $\Sigma_a \rightarrowtail \Sigma_a$. In order for the abstract transition rules to be consistent abstractions of the concrete transition rules, we impose that: $\forall \sigma_1, \sigma_2 \in \Sigma : \forall s_1, s_2 \in \Sigma_a : \sigma_1 \rightarrowtail \sigma_2 \wedge s_1 = \alpha(\sigma_1) \wedge s_1 \rightarrowtail s_2 \Rightarrow \sigma_2 \in \gamma(s_2)$.

The AbstractSimplify transition is a transition of the form $\Sigma_a \rightarrowtail$ answers where answers $= \texttt{one}(\Sigma_a) \mid \texttt{two}(\Sigma_a, \Sigma_a)$. Our framework requires that the AbstractSimplify transition satisfies the following constraint:

$$\begin{gathered}\forall \sigma_1, \sigma_2 \in \Sigma : \forall s_1, s_2 \in \Sigma_a : \\ \sigma_1 \rightarrowtail_{Simplify}\ \sigma_2 \quad \wedge \quad s_1 = \alpha(\sigma_1) \quad \wedge \quad s_1 \rightarrowtail \texttt{one}(s_2) \\ \Downarrow \\ \sigma_2 \in \gamma(s_2)\end{gathered}$$

$$\begin{gathered}\forall \sigma_1, \sigma_2 \in \Sigma : \forall s_1, s_2, s_3 \in \Sigma_a : \\ \sigma_1 \rightarrowtail_{Simplify}\ \sigma_2 \quad \wedge \quad s_1 = \alpha(\sigma_1) \quad \wedge \quad s_1 \rightarrowtail \texttt{two}(s_2, s_3) \\ \Downarrow \\ \sigma_2 \in \gamma(s_2) \vee \sigma_2 \in \gamma(s_3)\end{gathered}$$

For the other transition rules the program point of the resulting state σ_2 is completely determined by the program point of the original state σ_1. For the **Simplify** transition there are two possibilities for the resulting program point given the original program point $p/n : i$. If the active constraint is removed by the transition, the resulting program point is $\square$ and if it is not, the program point is $p/n : i + 1$.

The way multiple resulting states are combined by the framework is discussed below.

4.3 The Generic Abstract Semantics

Here we explain the generic semantics of the framework, based on the analysis-specific implementations of the abstract domain and abstract transition rules.

The concrete operational semantics specifies that a program starts from an initial state and transition rules are applied until a final state is reached. In the following we describe what initial state is used by the framework and how the final state is obtained by applying abstract transition rules. In particular the issues of non-determinism are discussed.

4.3.1 Generic Initial State

For any CHR program, an infinite number of concrete initial states are possible, namely any $\langle G, [], \emptyset, \emptyset, \emptyset \rangle_1$ with G any finite list of CHR constraints and built-in constraints.

This infinite number of initial states may lead to an infinite number of abstract states, depending on the definition of α. However, in the generic framework we avoid this potential blow-up of initial states by restricting the initial goal to be a single CHR constraint c.

The above restriction is not a strong restriction. It is always possible to encode a list of multiple goals $c_1, \ldots, c_n$ in this way. Namely one can introduce a fresh constraint c and a new simplification rule $c \Leftrightarrow c_1, \ldots, c_n$. This new c can then serve as the single initial goal.

Similarly, it is possible to encode arbitrary sequences of constraints, using random data generators. A random data generator is no more than a built-in function that returns a random value in some domain. For example a random sequence of a and b constraints, denoted $(a|b)*$ as a regular expression, may be encoded as follows:

```
c <=> random_element([1,2,3],X), c(X).
c(1) <=> a, c.
c(2) <=> b, c.
c(3) <=> true.
```

Here the predicate `random_element/2` returns in its second argument a random element from its first argument. The constraint c serves as the initial goal.

4.3.2 Transition Rule Application

The generic framework applies the abstract transition rules on an initial state until a final state is reached. For most abstract program states, only one abstract transition rule applies and hence the framework's task is straightforward.

The AbstractSimplify is an exception, as already mentioned in Section 4.2.

It is the framework's task to take the determinism into account and compute the appropriate results from the two alternate possibilities.

Consider an abstract state s_0 where the AbstractSimplify transition applies. If $s_0 \rightarrowtail_{AS} \texttt{one}(s_1)$ then s_1 is the resulting state. If $s_0 \rightarrowtail \texttt{two}(s_1, s_2)$ then there are two possible results. In order to find a least upper bound we must extend the states to final states and then build the least upper bound.

The framework then computes the following final state s_* for s_0:

$$s_* = \begin{cases} s_1 & \text{, if } s_0 \rightarrowtail_{AS} \texttt{one}(s_1) \\ s_1^* \sqcup s_2^* & \text{, if } s_0 \rightarrowtail_{AS} \texttt{two}(s_1, s_2) \\ & \quad \text{and } s_1 \rightarrowtail^* s_1^*, s_2 \rightarrowtail^* s_2^* \end{cases}$$

with $\rightarrowtail_{AS}$ an application of the AbstractSimplify Rule.

4.3.3 Non-determinism in the Simplify and Propagate Rules

While the above accounts for the non-determinism in simplification matching caused by abstraction, it does no account for the inherent non-determinism of these transitions in the concrete semantics.

Namely, for a simplification transition, if more than one combination of partner constraints are possible, the concrete semantics does not specify what particular combination is chosen. To account for this non-determinism the formulation of the AbstractSimplify transition should capture all possible concrete transitions. In particular, if for concrete state σ there are n different possible resulting final states $\sigma_1, \ldots, \sigma_n$, then $\alpha(\sigma) \rightarrowtail_{AS} \texttt{one}(s')$ or $\alpha(\sigma) \rightarrowtail_{AS} \texttt{two}(s', s'')$ such that $\bigsqcup_{i=1}^{n} \alpha(\sigma_i) \preceq s'$.

Similarly, for a propagation transition, multiple combination transitions are possible. In addition, for a propagation transition, multiple applications are possible in a sequence. However, the order of the sequence is not specified by the concrete semantics either. Hence, an abstract propagation transition has to capture all possible partner combinations and all possible sequences in which they are dealt with.

4.3.4 Non-determinism in the Solve Rule

The non-determinism inherent in the concrete Solve rule lies in the order of the triggered constraints as they are put on the execution stack: all possible orderings are allowed.

Hence, an abstract domain has to provide an abstraction that takes into account all possible orderings.

One approach would be, if the abstract domain permits, to compute the final state s_o for all $o \in O$ with O the set of all possible orderings and to combine these final states to a single final state s as follows: $s = \bigsqcup_{o \in O} s_o$.

However, this requires sufficiently concrete information about the number of triggered constraints in the abstract domain. Typically the abstract domain cannot provide any quantitative bound on the number of triggered constraints. Hence an infinite number of orderings are possible: all possible permutations of constraint sequences of any integer length.

A possible finite approximation of this infinite number of possibilities is to perform the following fixpoint computation. Say $\{c_i | 1 \leq i \leq n\}$ are all the possible distinct abstract CHR constraints to trigger. Then, starting from abstract state s_0, the final state s_f after triggering all constraints in any quantity is s_k, where:

$$s_j = \bigsqcup \{s_j^i | \mathsf{new\_goal}(s_{j-1}, c_i) \rightarrowtail^* s_j^i \wedge \mathsf{final}(s_j^i) \wedge 1 \leq i \leq n\}$$

for $j > 0$ and k is the smallest integer such that $s_k = s_{k+1}$. In the above formula $\mathsf{new\_goal}$ is the function that replaces the empty goal in a final abstract state with a new goal.

This generic approach is illustrated in the prototype groundness analysis, discussed in Section 6.

Due to its generality it may cause a huge loss of precision as well as an exponential number of intermediate states. Hence, in practice, better domain specific techniques should be studied.

For example, in the late storage analysis discussed in the next section, the worst possible abstract state is immediately obtained in the $\mathsf{AbstractSolve}$ transition, before triggered constraints are considered. Hence, there is no need to actually compute the triggering of constraints; the outcome is already determined. This avoids substantial needless overhead.

5. LATE STORAGE ANALYSIS

In this section we illustrate the use of the abstract interpretation framework for CHRs with a CHR-specific analysis: late storage. This analysis is useful in CHR compilers to drive several optimizations.

In Section 5.1 we define the property that the analysis tracks. Next, the abstract domain and transition rules of the analysis are defined in Sections 5.2 and 5.3 respectively. Section 5.4 illustrates the application on a small program.

5.1 The Observation Property

The aim of late storage analysis is to determine for an active CHR constraint whether it can be stored later rather than stored before its rules are searched for matching. The manner in which this is done is by determining when the first possible interaction will be with the active CHR constraint when executing one of its bodies.

In general it is better to store a constraint in the constraint store as late as possible. The reason is that if the constraint is deleted before it is actually stored, the overhead of insertion in and removal from the constraint store are avoided.

The refined operational semantics however dictates that a constraint is inserted in the constraint store immediately when it is at the top of the execution stack. We want to avoid this when it does not make a difference to the final state.

At the latest, a constraint that is not deleted, has to be stored after all the rules have been tried out. There are however reasons to store a constraint early. Namely, if a rule applies, the body may be able to observe whether the active constraint is in the constraint store or not. If the active constraint may be observed, the constraint needs to be in the constraint store. Otherwise it does not have to be in the constraint store, because its presence cannot impact the execution.

Definition 1 (Observed). *A constraint in the constraint store is observed, if it is triggered by a built-in constraint or if it acts as a partner constraint to an active constraint in a rule firing.*

To correctly define the analysis of "observation" as an abstract interpretation we have to extend the call-based operational semantics to make this visible. We will only be interested in finding the observed occurrences of constraints in the activation stack.

Denote an *observed* occurrence $c\#i : j$ by starring e.g. $c\#i : j^*$. Define

$$\begin{array}{rcll}
obs(c\#i:j) &=& c\#i:j^* & \\
obs(c\#i:j^*) &=& c\#i:j^* & \\
obs([], S) &=& [] & \\
obs([c\#i:j|G], S) &=& [obs(c\#i:j)|obs(G,S)] & c\#i \in S \\
obs([c\#i:j|G], S) &=& [c\#i:j|obs(G,S)] & c\#i \notin S
\end{array}$$

We only need to redefine the **Solve**, **Simplify** and **Propagate** rules slightly. Basically we modify the activation stack to record which constraints have been observed by any of these transitions.

1. Solve $\langle c, A, S, B, T\rangle_n \rightarrowtail_{So} \langle \square, A', S', B', T'\rangle_{n'}$ where c is a built-in constraint. If $\mathcal{D} \models \neg\bar{\exists}_\emptyset(c \wedge B)$, then $S' = S$, $B' = c \wedge B$, $T' = T$, $n' = n$. Otherwise ($\mathcal{D} \models \bar{\exists}_\emptyset(c \wedge B)$), where

$$\langle S_1, obs(A, S_1), S, c \wedge B, T\rangle_n \rightarrowtail^* \langle \square, A', S', B', T'\rangle_{n'}$$

and $S_1 = solve(S, B, c)$.

4. Simplify

$$\langle c\#i:j, A, \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail_{Si} \\ \langle \square, A', S', B', T''\rangle_{n'}$$

where

$$\langle C, obs(A, H_1 \cup H_2 \cup H_3), H_1 \uplus S, \theta \wedge B, T'\rangle_n \\ \rightarrowtail^* \langle \square, A', S', B', T''\rangle_{n'}$$

where the j^{th} occurrence of the CHR predicate of c in a (renamed apart) rule $r \in P$:

$$H_1' \setminus H_2', d_j, H_3' \iff g \mid C$$

and there exists matching substitution θ is such that $c = \theta(d_j)$, $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$, $chr(H_3) = \theta(H_3')$, and $\mathcal{D} \models B \rightarrow \bar{\exists}_{vars(r)}(\theta \wedge g)$, and the tuple $id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r] \notin T$. The substitution θ must also rename apart all variables appearing only in g and C. In the intermediate transition sequence $T' = T \cup \{id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r]\}$.

If so such matching substitution exists then

$$\langle c\#i:j, A, S, B, T\rangle_n \rightarrowtail_{Si} \langle c\#i:j+1, A, S, B, T\rangle_n$$

5. Propagate

$$\langle c\#i:j, A, \{c\#i\} \uplus S, B, T\rangle_n \rightarrowtail_P \langle G, A_k, S_k, B_k, T_k\rangle_{n_k}$$

where the j^{th} occurrence of the CHR predicate of c in a rule $r \in P$:

$$H_1', d_j, H_2' \setminus H_3' \iff g \mid C$$

Let $A_0 = A$, $S_0 = S \uplus \{c\#i\}, B_0 = B, T_0 = T, n_0 = n$.

Now assume, for $1 \leq l \leq k$ and $k \geq 0$, the series of transitions

$$\begin{array}{l}\langle C_l, [c\#i:j|obs(A_{l-1}, H_{1l} \cup H_{2l} \cup H_{3l})], \\ \qquad H_{1l} \uplus \{c\#i\} \uplus H_{2l} \uplus R_l, B_{l-1}, T_{l-1} \cup \{t_l\}\rangle_{n_{l-1}} \\ \rightarrowtail^* \langle \Box, [\_|A_l], S_l, B_l, T_l\rangle_{n_l}\end{array}$$

where $\{c\#i\} \uplus H_{1l} \uplus H_{2l} \uplus H_{3l} \uplus R_l = S_{l-1}$ and there exists a matching substitution θ_l such that

$$\begin{cases} c = \theta_l(d_j) \\ C_l = \theta_l(C) \\ chr(H_{1l}) = \theta_l(H_1') \\ chr(H_{2l}) = \theta_l(H_2') \\ chr(H_{3l}) = \theta_l(H_3') \\ \mathcal{D} \models B_{l-1} \rightarrow \bar{\exists}_{\theta_l(r)} \theta_l(g) \\ t_l = id(H_{1l}) +\!\!+ [i] +\!\!+ id(H_{2l}) +\!\!+ id(H_{3l}) +\!\!+ [r] \notin T_{l-1} \end{cases}$$

where θ_l renames apart all variables only appearing in g and C (separately for each l).

Furthermore, for $k + 1$ no such transition is possible.

The resulting goal G is either $G = \Box$ if $\mathcal{D} \models \neg\bar{\exists}_\emptyset B_k$ (i.e. failure occurred) or $G = c\#i:j + 1$ otherwise.

Example 1. *When examining the derivation shown in Section 3.3 the altered versions of the transitions above make one change. After the **Simplify** transition in the derivation for s, the p in the store is observed, so the new state is*

$$\rightarrowtail_{Si} \quad \langle \Box, [p\#1:4^*], \{q\#2\}, \emptyset, \bullet\rangle_4$$

5.2 Abstract Domain

The abstract state used for this analysis is rather simple. We abstract CHR constraints by their predicate names, and built-in constraints as simply the special predicate name `builtin`. The abstract state simple holds an abstraction of the goal or active constraint:occurrence, and an abstraction of the call stack A. The abstracted call stack is a set. It denotes the predicate occurrences which have not been observed.

Let c be a built-in constraint and p a CHR constraint, and S a set or multiset of CHR constraints. We define the late storage abstraction α_{ls} as follows:

$$\begin{array}{rcll} \alpha_{ls}(c) & = & \texttt{builtin} & , c \text{ built-in} \\ \alpha_{ls}(p(t_1, \ldots, t_n)) & = & p & \\ \alpha_{ls}(p(t_1, \ldots, t_n)\#i) & = & p & \\ \alpha_{ls}(p(t_1, \ldots, t_n)\#i:j) & = & p:j & \\ \alpha_{ls}([]) & = & [] & \\ \alpha_{ls}([c|G]) & = & [\alpha_{ls}(c)|\alpha_{ls}(G)] & \\ \alpha_{ls}(S) & = & \{\alpha_{ls}(c)|c \in S\} & , S \text{ set} \\ \alpha_{ls}(\langle G, A, \_, \_, \_\rangle_\_) & = & \langle \alpha_{ls}(G), \alpha_{ls}(\mathsf{unobserved}(A))\rangle & \end{array}$$

where unobserved is defined as

$$\mathsf{unobserved}(A) = \left\{ p \,\middle|\, \begin{array}{l} p(t_1, \ldots, t_n)\#i:j \in \mathsf{list2set}(A), \\ \neg\exists p(t_1', \ldots, t_n')\#i':j'^* \in \mathsf{list2set}(A) \end{array} \right\}$$

$$\mathsf{list2set}(A) = \begin{cases} \emptyset & , A = [] \\ \{a\} \cup \mathsf{list2set}(A') & , A = [a|A'] \end{cases}$$

Note we abstract built-in constraints, and non-identified CHR constraints by keeping the predicate. We abstract identified CHR constraints by removing the identity number and occurrenced identified CHR constraints just keeping track of the occurrence number. We eliminate observed constraints from the execution stack using the auxiliary function unobserved.

The abstracted call stack is a set. It denotes the predicates which have not been observed in the future computation.

The partial ordering on states is $\langle G, A\rangle \preceq_{ls} \langle G', A'\rangle$ iff $G = G'$ and $A' \subseteq A$. Clearly the abstract domain forms a lattice with the ordering relation $\preceq_{ls}$. The least upper bound operator $\sqcup_{ls}$ can be defined as follows:

$$\langle G, A_1\rangle \sqcup_{ls} \langle G, A_2\rangle = \langle G, (A_1 \cap A_2)\rangle$$

5.3 Abstract Transition Rules

Each abstract operation must provide two things:
(a) whether it is applicable at the current state s_0, and
(b) the resulting state afterwards s. We overload $\rightarrowtail^*$ to indicate the transitive reflexive closure of the abstract transitions.

5.3.1 AbstractSolve

$s_0 = \langle \texttt{builtin}, A\rangle \rightarrowtail_{AS} \langle \Box, \emptyset\rangle = s$ Applicable always when the goal is `builtin`.

A built-in constraint may possibly trigger any constraint in the constraint store. Hence all the constraints in the call stack are *possibly observed.*

For every constraint name c, the following subcomputation needs to be run to cover all execution paths, despite the fact that no information is carried over to s: $\langle c, \emptyset\rangle \rightarrowtail^* \langle \Box, \emptyset\rangle$.

Technically, the output state of one triggered constraint should become the input state of the next according to ω_c. Moreover, the constraints could be run in any order.

However, this computation is a safe approximation, since every initial and final state has a known empty A.

5.3.2 Abstract(Re)Activate

$s_0 = \langle c, A\rangle \rightarrowtail_{AA} \langle c:1, A\rangle = s$. Applicable if c is a non-occurrenced CHR constraint.

5.3.3 AbstractDrop

$s_0 = \langle c:j, A\rangle \rightarrowtail_{ADp} \langle \Box, A\rangle = s$. Applicable if no occurrence j exists for CHR predicate c.

5.3.4 AbstractGoal

$s_0 = \langle [c_{k_1}, \ldots, c_{k_n}], A\rangle \rightarrowtail_{AG} \langle \Box, A'\rangle = s$ where

$$\langle c_{k_i}, A\rangle \rightarrowtail^* \langle \Box, A_i\rangle$$

and $A' = \bigcap_{i=1}^n A_i$

Technically, the output state of one goal should become the input state of the next according to the call-based operational semantics. However, this definition here captures the meaning of *possibly observed* too: If a constraint in the call stack is possibly observed by any goal in a conjunction, it is possibly observed by the entire conjunction.

5.3.5 AbstractSimplify

$s_0 = \langle c\!:\!j, A_0\rangle$. Applicable if occurrence j is a simplification occurrence is a rule $r \in P$:

$$H'_1 \setminus H'_2, d_j, H'_3 \iff g \mid C$$

Let $O = \alpha_{ls}(H'_1 \cup H'_2 \cup H'_3)$ and let $A_1 = A_0 - O$.

Assume

$$\langle \alpha_{ls}(C), A_1\rangle \rightarrowtail^* \langle \Box, A_2\rangle$$

Then $s = \langle \Box, A_2\rangle$ and the result of the rule is:

- $\texttt{one}(s)$, if r is an unconditional simplification rule, i.e. of the form $c(\bar{x}) \Leftrightarrow C$ with all $x \in \bar{x}$ distinct variables. Namely, the rule application only fails when the active constraint is not in the constraint store, this leads to a state $\langle \Box, A_0\rangle$ which when lubbed with s gives s.
- $\texttt{two}(s, \langle c\!:\!j+1, A_0\rangle)$ otherwise.

In the first case, the rule must always fire, the exception is that the active constraint may have already been deleted. In this case we can terminate with no new observation since the active constraint cannot match further.

If the rule is not a unconditional simplification rule then we simply either succeeded and observed, or move to the next occurrence.

In fact we could just replace the second case by $\texttt{one}(\langle c : j+1, A_2\rangle)$ without loss of accuracy, but we give the more complicated definition to illustrate the use of `two`.

5.3.6 AbstractPropagate

$s_0 = \langle c\!:\!j, A_0\rangle$. Applicable if occurrence j is a propagation occurrence in rule $r \in P$:

$$H'_1, d_j, H'_2 \setminus H'_3 \iff g \mid C$$

Let $O = \alpha_{ls}(H'_1 \cup H'_2 \cup H'_3)$, $A_1 = A_0 - O$. Let $A_2 = A_1 \cup \{\alpha_{ls}(c)\}$.

Assume

$$\langle \alpha_{ls}(C), A_2\rangle \rightarrowtail^* \langle \Box, A_3\rangle$$

Let $A_4 = A_3 \setminus (\{\alpha_{ls}(c)\} \setminus A_1)$, removing $\alpha_{ls}(c)$ from the execution stack if it was not present initially.

Then the result of the rule is $\langle c\!:\!j+1, A_4\rangle$.

Note that the active constraint c may have been observed in C iff $c \notin A_3$.

Note here we treat the rule as if it always could have fired. This is clearly safe.

5.4 Example Analysis

Consider the execution of the goal `p` with respect to the following (numbered) CHR program

```
p1 ==> q.
p2, t1 <=> r.
p3, r1 ==> true.
p4 ==> s.
p5, s1 <=> true.
```

The example derivation is shown below. For simplification rules we show the states s_1 and s_2 in the answer $\texttt{two}(s_1, s_2)$ after lines labelled first and second, and then give the two derivations that lead to the lub.

$\quad \langle \texttt{p}, \emptyset\rangle$
$\rightarrowtail_{AA} \quad \langle \texttt{p}\!:\!1, \emptyset\rangle$
$\rightarrowtail_{AP} \quad \langle \texttt{p}\!:\!2, \emptyset\rangle \qquad \langle \texttt{q}, \{\texttt{p}\}\rangle \rightarrowtail^* \langle \Box, \{\texttt{p}\}\rangle$

first

$\quad \rightarrowtail_{ASi} \quad \langle \Box, \emptyset\rangle \qquad \langle \texttt{r}, \emptyset\rangle \rightarrowtail^* \langle \Box, \emptyset\rangle$

second

$\quad \rightarrowtail_{\neg ASi} \quad \langle \texttt{p}\!:\!3, \emptyset\rangle$
$\quad \rightarrowtail_{AP} \quad \langle \texttt{p}\!:\!4, \emptyset\rangle \qquad \langle \Box, \{\texttt{p}\}\rangle \rightarrowtail^* \langle \Box, \{\texttt{p}\}\rangle$
$\quad \rightarrowtail_{AP} \quad \langle \texttt{p}\!:\!5, \emptyset\rangle \qquad \langle \texttt{s}, \{\texttt{p}\}\rangle \rightarrowtail^* \langle \Box, \emptyset\rangle$

first

$\qquad \rightarrowtail_{ASi} \quad \langle \Box, \emptyset\rangle \qquad \langle \Box, \emptyset\rangle \rightarrowtail^* \langle \Box, \emptyset\rangle$

second

$\qquad \rightarrowtail_{\neg ASi} \langle \texttt{p}\!:\!6, \emptyset\rangle$
$\qquad \rightarrowtail_{ADp} \langle \Box, \emptyset\rangle$

lub

$\quad \rightarrowtail_{\sqcup} \quad \langle \Box, \emptyset\rangle$

lub

$\rightarrowtail_{\sqcup} \quad \langle \Box, \emptyset\rangle$

$\quad \langle \texttt{q}, \{\texttt{p}\}\rangle$
$\rightarrowtail_{AA} \quad \langle \texttt{q}\!:\!1, \{\texttt{p}\}\rangle$
$\rightarrowtail_{ADp} \quad \langle \Box, \{\texttt{p}\}\rangle$

$\quad \langle \texttt{r}, \emptyset\rangle$
$\rightarrowtail_{AA} \quad \langle \texttt{r}\!:\!1, \emptyset\rangle$
$\rightarrowtail_{AP} \quad \langle \texttt{r}\!:\!2, \emptyset\rangle \qquad \langle \Box, \{\texttt{r}\!:\!1\}\rangle \rightarrowtail^* \langle \Box, \{\texttt{r}\!:\!1\}\rangle$
$\rightarrowtail_{ADp} \quad \langle \Box, \emptyset\rangle$

$\quad \langle \texttt{s}, \{\texttt{p}\}\rangle$
$\rightarrowtail_{AA} \quad \langle \texttt{s}\!:\!1, \{\texttt{p}\}\rangle$

first

$\quad \rightarrowtail_{ASi} \quad \langle \Box, \emptyset\rangle \qquad \langle \Box, \emptyset\rangle \rightarrowtail^* \langle \Box, \emptyset\rangle$

second

$\quad \rightarrowtail_{\neg ASi} \quad \langle \texttt{s}\!:\!2, \{\texttt{p}\}\rangle$
$\quad \rightarrowtail_{ADp} \quad \langle \Box, \{\texttt{p}\}\rangle$

lub

$\rightarrowtail_{\sqcup} \quad \langle \Box, \emptyset\rangle$

Note that we observe the `p` only in the derivation for `s` hence we can safely delay storage of `p` until just before the execution of this body.

6. GROUNDNESS ANALYSIS

In this section we illustrate the use of the abstract interpretation framework by lifting the classical groundness analysis for Prolog to CHRs.

In the groundness analysis for CHRs we capture the groundness of variables at the scope of rules and arguments of constraints. Unlike Prolog we do not go as far as capturing groundness relations between all variables.

Sections 6.1 and 6.2 present the abstract domain and transition rules respectively. The analysis is illustrated by means of an example in Section 6.3.

6.1 Abstract Domain

In abstracting groundness properties of a CHR execution we will be interested in three parts of the concrete state, the goal, the CHR constraint store, and the built-in constraint store.

Groundness is not directly affected by CHR constraints, but only through built-in constraints of the underlying constraint domain $\mathcal{D}$. Hence, we assume that we have an abstract domain $\mathcal{P}$ for tracking groundness of the underlying constraint domain $\mathcal{D}$, providing the following:

- the operations $\alpha_{\mathcal{P}}, \preceq_{\mathcal{P}}, \sqcup_{\mathcal{P}}, \ldots$
- the abstract conjunction, denoted by $\wedge_{\mathcal{P}}$ joins two abstract descriptions
- the function $\mathsf{Aadd}_{\mathcal{P}}$ joins an abstract description with a concrete constraint
- the function $\mathsf{grounds}_{\mathcal{P}}(D)$, which returns the set of variables grounded by abstract description D
- the abstract projection function $\bar{\exists}^{\mathcal{P}}_V F$ which abstracts the projection $\bar{\exists}_V F$ the projection of F onto the variables V.

We abstract the state by an abstract goal, which only removes occurrence numbers, an abstract store which stores, for each CHR constraint, the least upper bound of the groundness descriptions of the CHR constraint instances in the store, and the abstract underlying store, which is just given using the domain $\mathcal{P}$ restricted to the variables in the goal.

$$\begin{array}{rcll}
\alpha_g(c) & = & c & c \text{ is built-in} \\
\alpha_g(p(t_1,\ldots,t_n)) & = & p(t_1,\ldots,t_n) & \\
\alpha_g(p(t_1,\ldots,t_n)\#i) & = & p(t_1,\ldots,t_n) & \\
\alpha_g(p(t_1,\ldots,t_n)\#i\!:\!j) & = & p(t_1,\ldots,t_n)\!:\!j &
\end{array}$$

$$\begin{array}{rcll}
\alpha_g([]) & = & [] & \\
\alpha_g([c|G]) & = & [\alpha_g(c)|\alpha_g(G)] & \\
\alpha_g(S) & = & \{\alpha_g(c) | c \in S\} & S \text{ set or multiset}
\end{array}$$

$$\alpha_g(p(t_1,\ldots,t_n)\#i, B) = p(x_1,\ldots,x_n) \leftarrow D$$
$$\text{where } D = \bar{\exists}^{\mathcal{P}}_{x_1,\ldots,x_n} \alpha_{\mathcal{P}}(B \wedge x_1 = t_1 \wedge \cdots \wedge x_n = t_n)$$

$$\alpha_g(S,B) = snf(\{\alpha_g(c,B) | c \in S\}) \qquad S \text{ set or multiset}$$
$$\alpha_g(\langle G, \_, S, B, \_\rangle_\_) = \langle \alpha_g(G), \alpha_g(S,B), \bar{\exists}^{\mathcal{P}}_{vars(G)} \alpha_{\mathcal{P}}(B)\rangle$$

where the function snf creates a normal form of the groundness description of the CHR constraint store, by ensuring there is at most one entry per CHR predicate. It is defined as follows:

$$\begin{array}{rcl}
snf(\emptyset) & = & \emptyset \\
snf(\{p(\bar{x}) \leftarrow D_1\} \uplus S) & = & snf(\{p(\bar{x}) \leftarrow D_1 \sqcup_{\mathcal{P}} D_2\} \uplus S') \\
& & \text{where } S = \{p(\bar{x}) \leftarrow D_2\} \uplus S' \\
snf(\{p(\bar{x}) \leftarrow D_1\} \uplus S) & = & \{p(\bar{x}) \leftarrow D_1\} \uplus snf(S), \\
& & \text{where } \neg\exists p(\bar{x}) \leftarrow D_2 \in S
\end{array}$$

We define pred as follows:

$$\begin{array}{rcl}
\mathsf{pred}(p(t_1,\ldots,t_n)) & = & p \\
\mathsf{pred}(p(x_1,\ldots,x_n) \leftarrow D) & = & p \\
\mathsf{pred}([]) & = & [] \\
\mathsf{pred}([c|G]) & = & [\mathsf{pred}(c)|\mathsf{pred}(G)]
\end{array}$$

The partial ordering $\preceq_g$ on states is

$$\langle G,S,B\rangle \preceq_g \langle G',S',B'\rangle \Leftrightarrow G = G' \wedge B \preceq_{\mathcal{P}} B' \wedge$$
$$(\forall p(\bar{x}) \leftarrow D \in S : \exists p(\bar{x}) \leftarrow D' \in S' : D \preceq_{\mathcal{P}} D')$$

Clearly the abstract domain forms a lattice with the ordering relation $\preceq_g$. The least upper bound operator $\sqcup_g$ can be defined as follows:

$$\langle G,S,B\rangle \sqcup_g \langle G,S',B'\rangle = \langle G, snf(S \cup S'), B \sqcup_{\mathcal{P}} B'\rangle$$

6.2 Abstract Transition Rules

Each abstract operation must provide two things:
(a) whether it is applicable at the current state s_0, and
(b) the resulting state afterwards s.

6.2.1 AbstractSolve

$s_0 = \langle c, S_a \uplus S_b, B\rangle$. Applicable when c is a built-in constraint. Define $S_a = \{p(\bar{x}) \leftarrow D \mid \bar{x} \subseteq \mathsf{grounds}_{\mathcal{P}}(D)\}$. Let $S_b = \{p_i(\bar{x}_i) \leftarrow D_i \mid 1 \leq i \leq n\}$.

Let

$$\begin{array}{rcl}
S_0 & = & S_a \uplus S_b \\
s_j & = & \langle \Box, S_j, \_\rangle = \sqcup_g\{s^i_j | \quad \langle p_i(\bar{x}_i), S_{j-1}, D_i\rangle \rightarrowtail^* s^i_j \wedge \\
& & \qquad \mathsf{final}(s^i_j) \wedge 1 \leq i \leq n\}, j \geq 1
\end{array}$$

and be k the smallest positive integer such that $s_k = s_{k-1}$. Then $s = \langle \Box, S_k, \mathsf{Aadd}_{\mathcal{P}}(c,B)\rangle$.

6.2.2 Abstract(Re)Activate

$s_0 = \langle c, S, B\rangle$. Applicable if c is a non-occurrenced CHR constraint. $s = \langle c\!:\!1, snf(\{\alpha_g(c,B)\} \cup S), B\rangle$.

6.2.3 AbstractDrop

$s_0 = \langle c\!:\!j, S, B\rangle$. Applicable if no occurrence j exists for CHR predicate c. $s = \langle \Box, S, B\rangle$.

6.2.4 AbstractGoal

$s_0 = \langle [c|G], S_0, B_0\rangle$. Let $B_1 = \bar{\exists}_{vars(c)} B_0$ and

$$\langle c, S_0, B_1\rangle \rightarrowtail^* \langle \Box, S, B_2\rangle$$

then $s = \langle G, S, B_0 \wedge_{\mathcal{P}} B_2\rangle$.

6.2.5 AbstractSimplify

$s_0 = \langle c\!:\!j, S, B\rangle$. Applicable if occurrence j is a simplification occurrence in rule $r \in P$:

$$H'_1 \setminus H'_2, d_j, H'_3 \iff g \mid C$$

where exists θ such that $c = \theta(d_j)$, $H_1 \cup H_2 \cup H_3 \subseteq S$, and $\mathsf{pred}(H_i) = \mathsf{pred}(H'_i), 1 \leq i \leq 3$.

Suppose

$$\begin{array}{rcl}
H_i & = & [p_{i1}(\bar{x}_{i1}) \leftarrow D_{i1}, \ldots, p_{in_i}(\bar{x}_{in_i}) \leftarrow D_{in_i}] \\
\theta(H'_i) & = & [p_{i1}(\bar{t}_{i1}), \ldots, p_{in_i}(\bar{t}_{in_i})]
\end{array}$$

Let

$$\begin{array}{rcl}
D_i & = & \mathsf{Aadd}(\wedge_{\mathcal{P}}\{D_{ij} \mid 1 \leq j \leq n_i\}, \wedge^{n_i}_{j=1}(\bar{x}_j = \bar{t}_j)) \\
D & = & \bar{\exists}_{vars(\theta(C))}\mathsf{Aadd}((D_1 \wedge_{\mathcal{P}} D_2 \wedge_{\mathcal{P}} D_3 \wedge_{\mathcal{P}} B), g)
\end{array}$$

Suppose that $\langle \theta(C), S, D\rangle \rightarrowtail^* \langle \Box, S', B'\rangle$.

Then $s = \langle \Box, S', B \wedge_{\mathcal{P}} \bar{\exists}_{vars(c)} B'\rangle$ and the result of the rule is:

- `one`$(s \sqcup_g \langle \Box, S, B\rangle)$, if r is an unconditional simplification rule, i.e. of the form $c(\bar{x}) \Leftrightarrow C$ with all $x \in \bar{x}$ distinct variables. The second state encodes the possibility that the active constraint has already been deleted.
- `two`$(s, \langle c\!:\!j+1, S, B\rangle)$ otherwise.

We find a possible match for each CHR constraint in the rule, assume that the guard holds, and determine the abstract underlying constraint store that must exist for the body of the rule from the matching. We execute the body of the rule with this store, without removing any constraints from the store (since we are not sure how many copies there are). The resulting abstract underlying store is projected back onto the active constraint and then added to the current store.

6.2.6 AbstractPropagate

$s_0 = \langle c{:}j, S, B\rangle$. Applicable if occurrence j is a propagation occurrence in rule $r \in P$:

$$H'_1, d_j, H'_2 \setminus H'_3 \iff g \mid C$$

where exists θ such that $c = \theta(d_j)$, $H_1 \cup H_2 \cup H_3 \subseteq S$ and $\mathsf{pred}(H_i) = \mathsf{pred}(H'_i), 1 \leq i \leq 3$.

Suppose

$$\begin{array}{rcl} H_i &=& [p_{i1}(\bar{x}_{i1}) \leftarrow D_{i1}, \ldots, p_{in_i}(\bar{x}_{in_i}) \leftarrow D_{in_i}] \\ \theta(H'_i) &=& [p_{i1}(\bar{t}_{i1}), \ldots, p_{in_i}(\bar{t}_{in_i})] \end{array}$$

Let

$$\begin{array}{rcl} D_i &=& \mathsf{Aadd}(\wedge_{\mathcal{P}}\{D_{ij} \mid 1 \leq j \leq n_i\}, \wedge_{j=1}^{n_i}(\bar{x}_j = \bar{t}_j)) \\ D &=& \bar{\exists}_{vars(\theta(C))}\mathsf{Aadd}((D_1 \wedge_{\mathcal{P}} D_2 \wedge_{\mathcal{P}} D_3 \wedge_{\mathcal{P}} B), g) \end{array}$$

Suppose that $\langle \theta(C), S, D\rangle \rightarrowtail^* \langle \Box, S', B'\rangle$. Then $s = \langle c : j+1, S', B \wedge_{\mathcal{P}} (\bar{\exists}_{vars(c)}B'\rangle$ is the result assuming the rule fired and $s_1 = \langle c{:}j+1, S, B\rangle$ and is the result if the rule did not fire the result of the rule is $\mathtt{one}(s \sqcup_g s_1)$.

6.3 Example Analysis

In this example analysis we will use the following simple abstract domain $\mathcal{P}$:

- $\alpha_{\mathcal{P}}(c) = \{x | x \in vars(c) \wedge c \rightarrow \mathtt{ground}(x)\}$
- $D_1 \preceq_{\mathcal{P}} D_2 \Leftrightarrow D_1 \supseteq D_2$
- $D_1 \sqcup_{\mathcal{P}} D_2 = D_1 \cap D_2$
- $D_1 \wedge_{\mathcal{P}} D_2 = D_1 \cup D_2$
- $\mathsf{Aadd}_{\mathcal{P}}(D, c) = D \cup \{x \in vars(c) | \exists D' \subseteq D : (\forall y \in D' : \mathtt{ground}(y)) \wedge c \rightarrow \mathtt{ground}(x)\}$
- $\mathsf{grounds}_{\mathcal{P}}(D) = D$
- $\bar{\exists}^{\mathcal{P}}_V D = D \cap V$

The example program we will analyze is `primes`, see [9], extended with an appropriate `main/0` constraint:

```
main₁ <=> N = 10, candidate(N).
candidate(N)₁ <=> N = 1 | true.
candidate(N)₂ <=> prime(N), M is N - 1, candidate(M).
prime(Y)₂ \ prime(X)₁ <=> 0 =:= X mod Y | true.
```

It computes the prime numbers between 1 and 10. The abstract derivation steps for the groundness analysis of this program are the following.

For brevity the abstract stores are shown separately: $S_1 = \{\mathtt{main}{:}\text{-}\emptyset\}$, $S_2 = S_1 \cup \{\mathtt{candidate}(N){:}\text{-}\{N\}\}$, $S_3 = S_2 \cup \{\mathtt{prime}(N){:}\text{-}\{N\}\}$.

$$\begin{array}{ll} & \langle \mathtt{main}, \emptyset, \emptyset\rangle \\ \rightarrowtail_{AA} & \langle \mathtt{main}{:}1, S_1, \emptyset\rangle \\ \rightarrowtail_{ASi} & \langle \Box, S_3, \emptyset\rangle \end{array}$$

$$\begin{array}{ll} & \langle [X = 10, \mathtt{candidate}(X)], S_1, \emptyset\rangle \\ \rightarrowtail_{AG} & \langle [\mathtt{candidate}(X)], S_1, \{X\}\rangle \\ \rightarrowtail_{AG} & \langle \Box, S_3, \{X\}\rangle \end{array}$$

$$\begin{array}{ll} & \langle X = 10, \emptyset, S_1\rangle \\ \rightarrowtail_{ASo} & \langle \Box, S_1, \{X\}\rangle \end{array}$$

$$\begin{array}{ll} & \langle \mathtt{candidate}(X), S_1, \{X\}\rangle \\ \rightarrowtail_{AA} & \langle \mathtt{candidate}(X){:}1, S_2, \{X\}\rangle \end{array}$$

first

$$\rightarrowtail_{ASi} \ \langle \Box, S_3, \{X\}\rangle$$

second

$$\rightarrowtail_{\neg ASi} \langle \Box, S_2, \{X\}\rangle$$

lub

$$\rightarrowtail_{\sqcup} \quad \langle \Box, S_3, \{X\}\rangle$$

$$\begin{array}{ll} & \langle [\mathtt{prime}(N), M \ \mathtt{is}\ N-1, \mathtt{candidate}(M)], S_2, \{N\}\rangle \\ \rightarrowtail_{AG} & \langle [M \ \mathtt{is}\ N-1, \mathtt{candidate}(M)], S_3, \{N, M\}\rangle \\ \rightarrowtail_{AG} & \langle [\mathtt{candidate}(M)], S_3, \{N, M\}\rangle \\ \rightarrowtail_{AG} & \langle \Box, S_3, \{N, M\}\rangle \end{array}$$

$$\begin{array}{ll} & \langle \mathtt{prime}(N), S_2, \{N\}\rangle \\ \rightarrowtail_{AA} & \langle \mathtt{prime}(N){:}1, S_3, \{N\}\rangle \end{array}$$

first

$$\rightarrowtail_{ASi} \ \langle \Box, S_3, \{N\}\rangle$$

second

$$\begin{array}{ll} \rightarrowtail_{\neg ASi} & \langle \mathtt{prime}(N){:}2, S_3, \{N\}\rangle \\ \rightarrowtail_{AP} & \langle \mathtt{prime}(N){:}3, S_3, \{N\}\rangle \\ \rightarrowtail_{ADp} & \langle \Box, S_3, \{N\}\rangle \end{array}$$

lub

$$\rightarrowtail_{\sqcup} \quad \langle \Box, S_3, \{N\}\rangle$$

We omit identical derivations for $[\mathtt{prime}(N), M \ \mathtt{is}\ N - 1, \mathtt{candidate}(M)]$ and $\mathtt{prime}(N)$ starting with CHR store S_3 rather than S_2. From this analysis we can conclude that the CHR constraints are ground at all times in this program.

7. IMPLEMENTATION AND EVALUATION

We have implemented both the late storage analysis and the groundness analysis in the hProlog version of the K.U. Leuven CHR system [11].

We have implemented the late storage and ground analyses to always start from an initial goal $\langle \mathtt{main}, \emptyset\rangle$ and $\langle \mathtt{main}, \emptyset, \emptyset\rangle$ respectively. The rules for the constraint `main/0` in a particular benchmark define all relevant call patterns for that benchmark.

7.1 Late Storage Analysis

The results of this analysis are used for optimization in our CHR compiler in the following way:

- The main philosophy in late storage is to delay constraint storage, so that some constraints are removed before they have to be stored. Those constraints then avoid the overhead of both storage and removal.

  The reference CHR implementation in SICStus [7] already has an approximate late storage optimization. Namely, it does not store an activated constraint straight away, but only ensures it is stored before a rule body of a propagation occurrence is executed. With this analysis, this optimization is improved: our compiler only ensures that an active constraint is stored before the execution of a body of a propagation occurrence, if the constraint may be observed during the execution of that body.

- For a particular class of constraints, our compiler derives that they are *never stored*. Never stored constraints are not stored before an unconditional simplification occurrence. An unconditional simplification occurrence, is an occurrence in a single-headed rule without any matching or guard. The following optimizations are possible for never stored constraints:

- A constraint that is never stored, cannot be triggered. Hence no checks are necessary to discern between activation and reactivation.
- A never stored constraint cannot be found in a constraint store. Hence if it occurs in a multi-headed rule, its partner constraints in that rule should not actively try to apply that rule, i.e. their occurrences are considered passive.
- A never stored constraint will not reconsider the same propagation rule twice with the same partner constraints. Hence no history needs to be maintained for that rule.

Hence, the code generated by our compiler is much closer to the code one would write for a deterministic procedure in the host language than for an arbitrary constraint without the never stored property.

In Table 1 we show the speed-ups caused by late storage analysis. For ten benchmarks, see [9], we compare immediate storage with the current implementation of the above optimizations that are enabled by late storage analysis. The timings for the optimized programs are given relative to those of the unoptimized programs.

Table 1: Late storage analysis: runtime results of optimized programs relative to unoptimized programs.

Benchmark	Optimized / Unoptimized
bool	17.6%
fib	72.3%
fibonacci	72.7%
leq	75.7%
mergesort	86.5%
primes	94.6%
uf	97.4%
uf_opt	106.5%
wfs	95.7%
zebra	89.1%

In Table 2 we show the number of dynamic constraint store insertions and deletions for these benchmarks. The number of insertions and and the number of deletions saved out by late storage analysis is of course identical. The considerable reduction of the `bool` benchmark is clearly explained by the drastic decrease in the number of operations. While even more operations have been saved out in the `leq` benchmark, the impact on its runtime is more modest, though still considerable. This is because the overall impact of these operations on the total runtime is less dominant.

7.2 Groundness Analysis

Our implementation of the groundness analysis uses the naive groundness domain for built-in constraints as it is described in Section 6.3.

Table 2: Late storage analysis: store operations without and with late storage.

Benchmark	Without		With	
	Insert	Delete	Insert	Delete
bool	359,996	359,996	8.33%	8.33%
fib	114,603	114,580	50.01%	50.00%
fibonacci	81,000	39,000	51.85%	0.00%
leq	34,280	34,280	5.16%	5.16%
mergesort	37,170	34,610	30.97%	25.86%
primes	4,999	4,632	49.99%	46.03%
uf	7,994	6,994	37.50%	28.57%
uf_opt	8,004	7,004	37.50%	28.57%
wfs	46,800	44,800	91.03%	90.62%
zebra	56,790	130,300	37.52%	28.60%

We use the derived groundness information in the following way. Our current implementation only performs optimizations for constraints that are always ground. This groundness information is supplied as groundness declarations by the programmer. In this case however, instead of programmer supplied declarations we derive the groundness declarations from the results of the groundness analysis and add them to the program. The compiler then takes these annotations into account as usual, and may perform the following optimizations:

- Hash tables (with $\mathcal{O}(1)$ lookup, insertion and deletion) may be used for indexing, if the lookup keys are ground. The keys need to be ground in order to always generate the same integer hash value throughout the program. This is not possible for variables and we have not explored any alternatives for them yet.
- No delay provisions have to be taken for constraints whose relevant arguments are ground. See [2, 10] for a discussion of delay avoidance. The following optimizations are possible for constraints that do not trigger:
  - No construction of a continuation goal to be executed on reactivation.
  - No redundant attempt to look for delay variables.
  - No history needs to be maintained for propagation rules that may be evaluated only once for a particular sequence of head constraints. Revisiting a propagation rule is possible when constraints triggers or one constraint observes the other before the latter has reached its occurrence in the propagation rule. Hence this optimization requires both input from the groundness and late storage analysis.

We have experimentally evaluated our groundness analysis on the seven benchmarks from the previous section that deal with ground constraints. Table 3 lists the runtime of the optimized benchmarks relative to the unoptimized ones.

It turns out that the annotations derived from the groundness analysis results are optimal for all but the `uf` and `uf_opt` benchmarks. Optimal means that the derived annotations are as strong as the actual calling patterns of the constrains in those benchmarks. For the `uf` and `uf_opt` the derived annotations are nearly optimal and the runtime performance

Table 3: Groundness analysis: runtime results for optimized programs relative to unoptimized programs.

Benchmark	Optimized/Unoptimized
fib	57.4%
fibonacci	57.8%
mergesort	11.0%
primes	81.0%
uf	2.6%
uf_opt	3.7%
wfs	73.3%

is not significantly different from optimal annotations. The results also show that even with a fairly weak groundness analysis for CHRs, fairly drastic speed-ups can be achieved for some programs.

8. CONCLUSION

To the best of our knowledge, this is the first work on using abstract interpretation for CHRs. Many ad-hoc analyses and optimizations have been developed for CHRs already: delay avoidance [2, 10], late storage, continuation optimization, and index optimization [2]. Typically the analysis processed to obtain the necessary information for these analyses is only discussed informally or left out altogether.

We have shown that it is possible to apply the general and structured ideas of abstract interpretation to CHRs. Based on our definition of the call-based refined operational semantics of CHRs, we have formulated a framework for abstract interpretation. To illustrate the framework we have formulated two analyses in it: the CHR specific late storage analysis and the groundness analysis which we have lifted to CHRs from logic programming. These examples show that it is possible to formally define program analyses for CHRs which yield useful information for program optimization.

8.1 Future Work

We have only presented rather straightforward analysis domains as an illustration of the framework. These analyses should of course be strengthened with additional control flow information, derived from other analyses. It is for example possible to derive from the late storage analysis the never stored property for some constraints. This information reduces the set of constraints that may be reactivated.

Moreover the groundness analysis has only been exploited in the case that arguments of constraints are ground throughout their full lifetime. Of course it is also possible to exploit the groundness information in other cases, i.e. when arguments are ground from a certain occurrence or at certain occurrences.

Many more analyses for CHRs should be considered within the framework as well as the combination of these analyses.

Several efficiency issues have risen during the formulation of our framework, namely the necessity for several fixpoint computations. The impact of these computations on the overall efficiency of analyses in our framework remains to be explored. Possibly widening strategies are necessary to avoid overly long analysis times for some domains. A comprehensive study of the time/accuracy trade-off is required.

The common abstract interpretation analysis technique may facilitate the more unified view of host language and CHRs to perform multi-language analysis. For example in the case of CHRs in Prolog, a single groundness analysis for both Prolog and its embedded CHR code seems required to obtain the strongest results since there is a reciprocal interaction between both languages. Probably a more unified semantics of both is necessary to accomplish this.

Acknowledgments

We would like to thank María García de la Banda and Bart Demoen for their help and useful contributions to this paper.

9. REFERENCES

[1] P. Cousot and R. Cousot. Abstract interpretation: a unifed lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th Symp. on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[2] G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. Extending arbitrary solvers with constraint handling rules. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and Practice of Declarative Programming*, pages 79–90. ACM Press, 2003.

[3] G. J. Duck, P. J. Stuckey, M. García 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.

[4] T. Frühwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Special Issue on Constraint Logic Programming*, volume 37, pages 95–138, October 1998.

[5] G. Duck and T. Schrijvers and P. Stuckey. An abstract interpretation framework for constraint handling rules. Report CW 391, Department of Computer Science, K.U.Leuven, Leuven, Belgium, Sept. 2004.

[6] C. Holzbaur and T. Frühwirth. Compiling constraint handling rules into Prolog with attributed variables. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in LNCS, pages 117–133. Springer Verlag, 1999.

[7] Intelligent Systems Laboratory. *SICStus Prolog User's Manual.* PO Box 1263, SE-164 29 Kista, Sweden, October 2003.

[8] K. Marriott, H. Søndergaard, and N. Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

[9] T. Schrijvers. CHR benchmarks and programs, October 2004. Available at http://www.cs.kuleuven.ac.be/˜toms/Research/CHR/.

[10] T. Schrijvers and B. Demoen. Antimonotony-based delay avoidance for CHR. Report CW 385, K.U.Leuven, Department of Computer Science, July 2004.

[11] T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: implementation and application. In *First Workshop on Constraint Handling Rules: Selected Contributions*, number 2004-01, 2004. ISSN 0939-5091.