An Abstract Interpretation Framework for Constraint Handling Rules

Gregory J. Duck *Tom Schrijvers*
Peter J. Stuckey

Report CW 391, September 2004

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

An Abstract Interpretation Framework for Constraint Handling Rules

Gregory J. Duck *Tom Schrijvers*
Peter J. Stuckey

Report CW 391, September 2004

Department of Computer Science, K.U.Leuven

Abstract

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

In this work we bring the general program analysis methodology of abstract interpretation to CHR: we formulate an abstract interpretation framework over the call-based operational semantics of CHR. 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.

Keywords : Constraint Handling Rules, abstract interpretation, program analysis, late storage, groundness.
CR Subject Classification : D.3.2, D.3.3, D.3.4

Abstract Interpretation for Constraint Handling Rules

Gregory J. Duck[1], Tom Schrijvers[2⋆], and Peter J. Stuckey[1,3]

[1] Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
{gjd,pjs}@cs.mu.oz.au
[2] Department of Computer Science
Katholieke Universiteit Leuven, Belgium
tom.schrijvers@cs.kuleuven.ac.be
[3] NICTA Victoria Laboratory

Abstract. Program analysis is essential for the optimized compilation of Constraint Handling Rules (CHR) as well as the inference of behavioral properties such as confluence and termination. Up to now all program analyses for CHR have been developed in an ad hoc fashion.
In this work we bring the general program analysis methodology of abstract interpretation to CHR: we formulate an abstract interpretation framework over the call-based operational semantics of CHR. 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.

1 Introduction

Constraint Handling Rules (CHRs) [Frü98] are a rule-based language developed for expressing constraint solvers.

Although the language is around for several years and has a reasonable reference implementation in SICStus Prolog [Int03], the number of people involved in optimized compilation of and program analysis for CHR has been limited until the recent appearance of new CHR systems [DSGH03,SD04a]. The need to communicate and compare between different CHR systems has given rise to the formulation of the more deterministic refined operational semantics [DSGH04] 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

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

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 [CC77] 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.

The rest of this paper is structured as follows. First, in Section 2, we briefly introduce CHR and its relevant concepts. Second, Section 3 presents the call-based refined operational semantics of CHR 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 CHR. For a more thorough overview of CHR we refer the reader to [Frü98].

2.1 CHR 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 CHR.

```
X leq X <=> true.
X leq Y <=> number(X), number(Y) | X =< Y.
X leq Y, Y leq X <=> X = Y.
X leq Y \ X leq Y <=> true.
X leq Y, Y leq Z ==> X leq 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 `X leq 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 body of the rule is only executed 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 `X leq Y` with a simple builtin 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 posed again in the body. However it is more efficient in that it never removes those head constraints and does not unnecessarily trigger 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 [Frü98] were essentially a multiset rewriting system. In [DSGH04] these semantics are defined as a transition system ω_t.

The semantics ω_t are 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 builtin or CHR constraint to add to the store at any stage is open.

Although these ω_t semantics are highly non-deterministic, actual CHR compilers typically already resolve most of this non-determinism statically. In fact, the refined operational semantics ω_r [DSGH04] reflect a large part of the increased determinism that is present in most CHR compilers we are aware of.

The ω_r semantics are more involved, yet they no longer leave any non-determinism in what transition rule should be applied in what state. Furthermore, they limit 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 builtin constraint to the store, and the order of that 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. They are a variant of the refined operational semantics ω_r [DSGH04] designed to make the analysis more straightforward. For the analysis of logic programs, we do

not directly analyses 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. [MSJ94]). We introduce the call-based refined operational semantics for CHRs for the same reason. We show in the appendix 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 are much closer to the procedure-based target languages, like Prolog and HAL, the current CHR compilers.

We believe 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.

The (call-based) refined semantics for CHRs use 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. This leads us to the keep track of occurrences of predicates. We assume that each constraint occurring in a rule is numbered from 1, in a top-down right-to-left manner. The numbered version of the `leq` program is

```
X leq1 X <=> true.
X leq Y2 <=> number(X), number(Y) | X =< Y.
X leq Y4, Y leq X3 <=> X = Y.
X leq Y6 \ X leq Y5 <=> true.
X leq Y8, Y leq Z7 ==> X leq Z.
```

We believe 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.

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, builtin 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 builtin constraints.

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 that only matches with occurrence j of constraint c should be considered when the constraint is active. The *execution stack* A is a sequence of constraints, identified CHR constraints and occurrenced identified CHR constraints, with a strict ordering in which only the top-most constraint is active.

The *builtin constraint store* B contains any builtin constraint that has been passed to 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 $\langle G, [], \emptyset, \emptyset, \emptyset \rangle_1$.

3.2 Transition Rules of ω_c

Execution proceeds by exhaustively applying transitions to the initial execution state until the builtin 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 $\rightarrowtail$.

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 builtin 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

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

in P and a substitution θ such that

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

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

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

where

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

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

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

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 $\mathcal{D} \models B \rightarrow \bar{\exists}_r(\theta \wedge g)$, and the tuple $id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r] \notin T$. 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 in a rule in P is

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

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

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

where $\{c\#i\} \uplus H_{1l} \uplus H_{2l} \uplus H_{3l} \trianglelefteq S_{l-1}$ 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 $\mathcal{D} \models \bar{\exists}_{\emptyset}(\neg B_k)$ (i.e. failure occurred) or $G = c\#i : j$ 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.

---

7. 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 builtin 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. □

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.

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

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

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

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

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

4 Abstract Interpretation Framework

In this section we present our generic abstract interpretation framework for CHR. The framework for CHR 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 provide 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 complete 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 concretization function γ from an abstract state to a set of concrete states.[4]

In order to be able to associate analysis information back to the program we also require the following restriction to the abstract state in the framework. First, it should be possible to determine from the abstract state whether it is a final state, i.e. with an empty goal $\Box$.

Second, it should be possible to determine from an abstract state the corresponding *program point*. We assume the predicate $\mathsf{pp}(s)$ returns predicate and occurrence for an active CHR constraint, or the sequence of predicates appearing in the corresponding concrete state, or an empty sequence indicating a final state. 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 ($\mathsf{pp}(s_1) = \mathsf{pp}(s_2)$). The demand for accurate program point information will complicate the abstract semantics somewhat, but result in more accurate analyses.

Some Guidelines

To be useful an abstracted state does not need to keep track of all possible components. It may assume the weakest possible information for any component. In our examples we shall mostly restrict ourselves to one or two components of the state.

Because there are multiple components, often individual components will themselves be described by abstract domains.

4.2 Abstract Transition Rules

The abstract domain must provide the following abstract operations:

- AbstractSolve,
- AbstractActivate,
- AbstractReactivate,
- AbstractDrop,
- AbstractSimplify,
- AbstractPropagate,
- AbstractDefault,
- AbstractGoal.

These abstract operations are abstractions of the transition rules defined by the call-based refined operational semantics of CHR, as given in Section 3.2.

Just as the concrete rules are transitions of the form $\Sigma \rightarrowtail \Sigma$, the abstract rules are transitions of the form $\Sigma_a \rightarrowtail \Sigma_a$. Also similar for the abstract transition rules is that their applicability to abstract states should be a partitioning of Σ_a.

The one exception to the form of the transition rules is AbstractSimplify. The Simplify rule is the only rule where the program point information may be

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

different depending on the workings of the rule: if there is a match it will be a final state, if not it will be the next occurrence. For the abstract operation it may not be possible to verify if there is a match or not, since the abstraction is too high, in this case the applicability can non-deterministically end in states for two different program points.

Hence this abstract transition rule should be of the form $\Sigma_a \rightarrowtail \mathsf{answers}(\Sigma_a)$, where $\mathsf{answers}$ is defined as follows:

$$\mathsf{answers}(\Sigma_a) = \mathtt{one}(\Sigma_a) \mid \mathtt{two}(\Sigma_a, \Sigma_a)$$

Its meaning is clear, there are either one or two possible resulting states.

- If the result is $\mathtt{one}(s_1)$, then the abstract state afterwards is s_1.
- If the result is $\mathtt{two}(s_1, s_2)$, then the abstract state afterwards is one of two states s_1 and s_2 where $\mathsf{pp}(s_1) \neq \mathsf{pp}(s_2)$.

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 specify 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.

Generic Initial State For any CHR program, an infinite number of concrete initial states are possible, namely any $\langle G, [], \emptyset, \emptyset, \emptyset \rangle_0$ with G any finite list of CHR constraints and builtin 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 builtin 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.

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 **??**.

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.

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 do not specify what particular combination is chosen. To account for this non-determinism the formulation of the AbstractSimplify transition should capture all possible. In particular, if for concrete state σ there n different possible resulting states $\sigma_1, \ldots, \sigma_n$, then $\alpha(\sigma) \rightarrowtail_{AS} \texttt{definitely}(s_s)$ or $\alpha(\sigma) \rightarrowtail_{AS} \texttt{maybe}(s_s, \_)$ such that $\bigsqcup_{i=1}^{n} \alpha(\sigma_i) \preceq s_s$.

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 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.

Non-determinism in the Solve Rule The non-determinism inherent in the concrete Solve rule is 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 each possible ordering o and to combine these final states to a single final state s as follows: $s = \bigsqcup_o s_o$.

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 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 `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 CHR 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 dictate 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 builtin constraint or if it serves as a partner constraint to an active constraint.*

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

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

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

$$\langle c, A, S, B, T\rangle_n \rightarrowtail_{So} \langle \Box, A', S', B', T'\rangle_{n'}$$

where c is a builtin 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

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

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

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

in P and a substitution θ such that

$$\begin{cases} ids(H_1) = \theta(H_1') \\ ids(H_2) = \theta(H_2') \\ \mathcal{D} \not\models B \rightarrow \exists_r(\theta \wedge g) \\ \mathcal{D} \models B \wedge c \rightarrow \exists_r(\theta \wedge g) \end{cases}$$

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.

---

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

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

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

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

and there exists matching substitution θ is such that $c = \theta(d_j)$, $ids(H_1) = \theta(H_1')$, $ids(H_2) = \theta(H_2')$, $ids(H_3) = \theta(H_3')$, and $\mathcal{D} \models B \rightarrow \bar{\exists}_r(\theta \wedge g)$, 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 in a (renamed apart) rule in P is

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

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

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

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

$$\begin{cases} c = \theta_l(d_j) \\ C_l = \theta_l(C) \\ chr(H_{1l}) = \theta_l(H'_1) \\ chr(H_{2l}) = \theta_l(H'_2) \\ chr(H_{3l}) = \theta_l(H'_3) \\ \mathcal{D} \models B_{l-1} \rightarrow \bar{\exists}_{\theta_l(r)} \theta_l(g) \\ t_l = id(H_{1l}) +\!\!+ [i] +\!\!+ id(H_{2l}) +\!\!+ id(H_{3l}) +\!\!+ [r] \notin T_{l-1} \end{cases}$$

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 $\mathcal{D} \models \bar{\exists}_{\emptyset}(\neg B_k)$ (i.e. failure occurred) or $G = c\#i : j+1$ otherwise. The resulting activation stack A' is the tail of A_k, i.e. $A_k = [\_|A']$.

Example 1. 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

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

5.2 Abstract Domain

The abstract state used for this analysis is rather simple. We abstract CHR constraints by their predicate names, and builtin constraints as simply a 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.

$$\begin{aligned}
\alpha_{ls}(c) &= \texttt{builtin} && c \text{ is builtin} \\
\alpha_{ls}(p(t_1, \ldots, t_n)) &= p && p \text{ is a CHR predicate} \\
\alpha_{ls}(p(t_1, \ldots, t_n)\#i) &= p && p \text{ is a CHR predicate} \\
\alpha_{ls}(p(t_1, \ldots, t_n)\#i : j) &= p : j && p \text{ is a CHR predicate} \\
\alpha_{ls}([]) &= [] \\
\alpha_{ls}([c|G]) &= [\alpha_{ls}(c)|\alpha_{ls}(G)] \\
\alpha_{ls}(S) &= \{\alpha_{ls}(c) | c \in S\} && S \text{ is a set or multiset of CHR constraints} \\
\alpha_{ls}(\langle G, A, _, _, _\rangle_{_}) &= \langle \alpha_{ls}(G), \alpha_{ls}(\mathsf{unobserved}(A))\rangle
\end{aligned}$$

where unobserved is defined as

$$\mathsf{unobserved}(A) = \{p \mid p(t_1,\ldots,t_n)\#i : j \in \mathsf{list2set}(A), \neg\exists p(t'_1,\ldots,t'_n)\#i' : j'^{*} \in \mathsf{list2set}(A)\}$$

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

and $\mathsf{pred}(p(t_1,\ldots,t_n)) = p$.

Note 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.

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

A builtin 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.

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.

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.

AbstractDefault $s_0 = \langle c : j, A\rangle \rightarrowtail_{ADf} \langle c : j+1, A\rangle = s$. Applicable if occurrence j exists for CHR predicate c.

AbstractGoal $s_0 = \langle [c_1, \ldots, c_n], A\rangle \rightarrowtail_{AG} \langle \Box, A'\rangle = s$ where

$$\langle c_i, A\rangle \rightarrowtail^* \langle \Box, A_i\rangle$$

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.

AbstractSimplify

$$s_0 = \langle c : j, A_0\rangle$$

Applicable if occurrence j is a simplification occurrence

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

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`.

AbstractPropagate

$$s_0 = \langle c : j, A_0\rangle$$

Applicable if occurrence j is a propagation occurrence

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

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 $A_4 = A_3 \setminus (\{\alpha_{ls}(c)\} \setminus A_1)$. 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 $c \notin A_3$. Also note that 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.

$$
\begin{array}{lll}
 & \langle p, \emptyset\rangle & \\
\rightarrowtail_{AA} & \langle p:1, \emptyset\rangle & \\
\rightarrowtail_{AP} & \langle p:2, \emptyset\rangle & \langle q, \{p\}\rangle \rightarrowtail^* \langle \Box, \{p\}\rangle \\
\multicolumn{3}{l}{\hrulefill \text{ first}} \\
\rightarrowtail_{ASi} & \langle \Box, \emptyset\rangle & \langle r, \emptyset\rangle \rightarrowtail^* \langle \Box, \emptyset\rangle \\
\multicolumn{3}{l}{\hrulefill \text{ second}} \\
\rightarrowtail_{\neg ASi} & \langle p:3, \emptyset\rangle & \\
\rightarrowtail_{AP} & \langle p:4, \emptyset\rangle & \langle \Box, \{p\}\rangle \rightarrowtail^* \langle \Box, \{p\}\rangle \\
\rightarrowtail_{AP} & \langle p:5, \emptyset\rangle & \langle s, \{p\}\rangle \rightarrowtail^* \langle \Box, \emptyset\rangle \\
 & \multicolumn{2}{l}{\hrulefill \text{ first}} \\
 & \rightarrowtail_{ASi} \langle \Box, \emptyset\rangle & \langle \Box, \emptyset\rangle \rightarrowtail^* \langle \Box, \emptyset\rangle \\
 & \multicolumn{2}{l}{\hrulefill \text{ second}} \\
 & \rightarrowtail_{\neg ASi} \langle p:6, \emptyset\rangle & \\
 & \rightarrowtail_{ADp} \langle \Box, \emptyset\rangle & \\
 & \multicolumn{2}{l}{\hrulefill \text{ lub}} \\
\rightarrowtail_{\sqcup} & \langle \Box, \emptyset\rangle & \\
\multicolumn{3}{l}{\hrulefill \text{ lub}} \\
\rightarrowtail_{\sqcup} & \langle \Box, \emptyset\rangle &
\end{array}
$$

$$
\begin{array}{ll}
 & \langle q, \{p\}\rangle \\
\rightarrowtail_{AA} & \langle q:1, \{p\}\rangle \\
\rightarrowtail_{ADp} & \langle \Box, \{p\}\rangle
\end{array}
$$

$$
\begin{array}{lll}
 & \langle r, \emptyset\rangle & \\
\rightarrowtail_{AA} & \langle r:1, \emptyset\rangle & \\
\rightarrowtail_{AP} & \langle r:2, \emptyset\rangle & \langle \Box, \{r:1\}\rangle \rightarrowtail^* \langle \Box, \{r:1\}\rangle \\
\rightarrowtail_{ADp} & \langle \Box, \emptyset\rangle &
\end{array}
$$

$$
\begin{array}{ll}
 & \langle s, \{p\}\rangle \\
\rightarrowtail_{AA} & \langle s:1, \{p\}\rangle
\end{array}
$$

$$\dfrac{\dfrac{}{\rightarrowtail_{ASi} \quad \langle \square, \emptyset \rangle \qquad \langle \square, \emptyset \rangle \rightarrowtail^* \langle \square, \emptyset \rangle}\ \text{first}}{\dfrac{\rightarrowtail_{\neg ASi}\ \langle s:2, \{p\} \rangle \quad \rightarrowtail_{ADp}\ \langle \square, \{p\} \rangle}{\rightarrowtail_{\sqcup} \quad \langle \square, \emptyset \rangle}\ \text{lub}}\ \text{second}$$

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 CHR.

In the groundness analysis for CHR 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 builtin 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

Note that $\bar{\exists}_V F$ is the projection of F onto the variables V, and similarly for the abstract version $\bar{\exists}^{\mathcal{P}}_V F$.

We abstract the state by an abstract goal, which only removes occurrence numbers, an abstract store which stores per constraints 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.

$$
\begin{aligned}
\alpha_g(c) &= c && c \text{ is builtin}\\
\alpha_g(p(t_1,\ldots,t_n)) &= p(t_1,\ldots,t_n) && p \text{ is a CHR predicate}\\
\alpha_g(p(t_1,\ldots,t_n)\#i) &= p(t_1,\ldots,t_n) && p \text{ is a CHR predicate}\\
\alpha_g(p(t_1,\ldots,t_n)\#i:j) &= p(t_1,\ldots,t_n):j && p \text{ is a CHR predicate}
\end{aligned}
$$

$$
\begin{aligned}
\alpha_g([]) &= []\\
\alpha_g([c|G]) &= [\alpha_g(c)|\alpha_g(G)]\\
\alpha_g(S) &= \{\alpha_g(c)|c\in S\} && S \text{ is a set or multiset of CHR constraints}
\end{aligned}
$$

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

$$
\begin{aligned}
\alpha_g(S,B) &= snf(\{\alpha_g(c,B)|c\in S\}) && S \text{ is a set or multiset of CHR constraints}\\
\alpha_g(\langle G,\_,S,B,\_\rangle_\_) &= \langle\alpha_g(G),\alpha_g(S,B),\bar{\exists}^{\mathcal{P}}_{vars(G)}\alpha_{\mathcal{P}}(B)\rangle
\end{aligned}
$$

where

$$
\begin{aligned}
snf(\emptyset) &= \emptyset\\
snf(\{p(\bar{x})\leftarrow D_1\}\cup S) &= \{p(\bar{x})\leftarrow D_1\sqcup_{\mathcal{P}} D_2\}\cup S' && ,\bar{x}\not\subseteq \mathsf{grounds}_{\mathcal{P}}(D_1)\wedge snf(S)=\{p(\bar{x})\leftarrow D_2\}\cup S'\\
snf(\{p(\bar{x})\leftarrow D_1\}\cup S) &= \{p(\bar{x})\leftarrow D_1\}\cup S' && ,\bar{x}\not\subseteq \mathsf{grounds}_{\mathcal{P}}(D_1)\wedge snf(S)\neq\{p(\bar{x})\leftarrow D_2\}\cup S'\\
snf(\{p(\bar{x})\leftarrow D_1\}\cup S) &= snf(S) && ,\bar{x}\subseteq \mathsf{grounds}_{\mathcal{P}}(D_1)
\end{aligned}
$$

We extend pred as follows:

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

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:

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

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.

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

Let

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

and be k the smallest positive integer such that $s_k=s_{k-1}$.

Then $s=\langle\Box,S_k,\mathsf{Aadd}_{\mathcal{P}}(c,B)\rangle$.

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

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

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

AbstractSimplify $s_0 = \langle c : j, H_1 \cup H_2 \cup H_3 \cup S, B\rangle$. Applicable if occurrence j is a simplification occurrence

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

where exists θ such that $c = \theta(d_j)$ and $\mathsf{pred}(H_i) = \mathsf{pred}(H_i'), 1 \leq i \leq 3$.

Suppose

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

Let

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

Suppose that

$$\langle \theta(C), H_1 \cup H_2 \cup H_3 \cup S, D\rangle \rightarrowtail^* \langle \Box, S', B'\rangle$$

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

- $\mathtt{maybe}(s, \langle \Box, H_1 \cup H_2 \cup H_3 \cup S, B\rangle)$, 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.
- $\mathtt{maybe}(s, s_0)$ 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.

AbstractPropagate $s_0 = \langle c : j, H_1 \cup H_2 \cup H_3 \cup S, B\rangle$. Applicable if occurrence j is a propagation occurrence

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

where exists θ such that $c = \theta(d_j)$ and $\mathsf{pred}(H_i) = \mathsf{pred}(H_i'), 1 \leq i \leq 3$.

Suppose

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

Let

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

Suppose that

$$\langle \theta(C), H_1 \cup H_2 \cup H_3 \cup S, D\rangle \rightarrowtail^* \langle \Box, S', B'\rangle$$

Then $s = \langle c : j, S', B \wedge_{\mathcal{P}} (\bar{\exists}_{vars(c)} B'\rangle$ and the result of the rule is $\mathtt{maybe}(s, s_0)$.

6.3 Example Analysis

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

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

The example program we will analyze is `primes`, see [Sch04], 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.

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

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

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

$$
\begin{array}{ll}
 & \langle candidate(X), \emptyset, \{X\} \rangle \\
\rightarrowtail_{AA} & \langle candidate(X) : 1, \emptyset, \{X\} \rangle \\
 & \quad \rightarrowtail_{ASi} \ \langle \Box, \emptyset, \{X\} \rangle \\
 & \\
 & \quad \rightarrowtail_{ADf} \ \langle candidate(X) : 2, \emptyset, \{X\} \rangle \\
 & \quad \rightarrowtail_{ASi} \ \langle \Box, \emptyset, \{X\} \rangle \\
\rightarrowtail_{\sqcup} & \langle \Box, \emptyset, \{X\} \rangle
\end{array}
$$

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

$$
\begin{array}{ll}
 & \langle prime(N), \emptyset, \{N\} \rangle \\
\rightarrowtail_{AA} & \langle prime(N) : 1, \emptyset, \{N\} \rangle \\
 & \quad \rightarrowtail_{ASi} \ \langle \Box, \emptyset, \{N\} \rangle \\
 & \\
 & \quad \rightarrowtail_{ADf} \ \langle prime(N) : 2, \emptyset, \{N\} \rangle \\
 & \qquad \rightarrowtail_{AP} \ \langle prime(N) : 2, \emptyset, \{N\} \rangle \\
 & \quad \rightarrowtail_{ADf} \ \langle prime(N) : 3, \emptyset, \{N\} \rangle \\
 & \quad \rightarrowtail_{ADr} \ \langle \Box, \emptyset, \{N\} \rangle \\
\rightarrowtail_{\sqcup} & \langle \Box, \emptyset, \{N\} \rangle
\end{array}
$$

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 K.U.Leuven CHR system [SD04a].

We have implemented the late storage and ground analyses to always start from an initial goal $\langle main, \emptyset \rangle$ and $\langle main, \emptyset, \emptyset \rangle$ 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 [Int03] 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 eight benchmarks, see [Sch04], we compare immediate storage with the current implementation of the above optimizations that are enabled by late storage analysis. The timings for immediate storage are given in milliseconds and the timings for the optimized programs relative to that.

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 builtin constraints as it is described in Section 6.3.

Benchmark	Without	With
bool	4,790	17.3%
fib	1,850	73.5%
fibonacci	1,570	61.8%
leq	1,630	70.6%
primes	1,170	93.2%
ta	1,700	96.5%
wfs	970	93.8%
zebra	1,930	85.5%
average	-	74.0%

Table 1. Late storage analysis: runtime results without and with late storage.

Benchmark	Without		With	
	Insert	Delete	Insert	Delete
bool	359,996	359,996	8.3 %	8.3 %
fib	114,603	114,580	50.0 %	50.0 %
fibonacci	121,500	58,500	51.9 %	0.0 %
leq	34,280	34,280	5.2 %	5.2 %
primes	4,999	4,632	50.0 %	46.0 %
ta	31,200	20,400	68.3 %	51.5 %
wfs	61,000	56,000	91.8 %	92.9 %
zebra	56,790	130,300	55.9 %	80.8 %

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

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 [DSGH03,SD04b] 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. Re-

visiting 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 five CHR benchmarks from [Sch04] where some constraint arguments are ground: `fib`, `fibonacci`, `primes`, `ta` and `wfs`. To each of these benchmarks we have added a `main/0` constraint representative of the use of the particular benchmark.

Table 3 lists the runtime of the above benchmarks, without and with the annotations inferred by the analysis. In addition, we list the results for the optimal hand annotations. The runtimes without annotations are in ms. The runtimes with annotations are relative to those without.

Benchmark	Without	With
fib	1,890	61.9%
fibonacci	1,580	58.2%
primes	1,600	58.1%
ta	2,850	21.1%
wfs	1,050	87.6%
average	-	57.4%

Table 3. Groundness analysis: runtime results without and with inferred annotations.

It turns out that the annotations derived from the groundness analysis results are optimal, i.e. they are as strong as the actual calling patterns of the constrains in those benchmarks. The results also show that even with a fairly weak groundness analysis for CHR, fairly good 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 CHR. Many ad-hoc analyses and optimizations have been developed for CHR already: ordinary and anti-monotonic delay avoidance [DSGH03,SD04b], index analysis and optimization [DSGH03], ... 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 CHR. Based on our definition of the call-based refined operational semantics of CHR, 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 CHR from logic programming. These two domains show that it is

possible to precisely and formally state program analyses in CHR 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.

Of course, many more analyses for CHR 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. It remains to be explored how much the impact of these computations is on the overall efficiency of analyses in our framework. 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.

A possibility for accuracy improvement across analysis domains is to use the more specialized operational semantics that are used by that compiler. The semantics of the compiler will typically be a more deterministic instance of the call-based semantics.

The common abstract interpretation analysis technique may facilitate the more unified view of host language and CHR to perform multi-language analysis. For example in the case of CHR in Prolog, a single groundness analysis for both the Prolog code 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 semantics 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.

References

[CC77] 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.

[DSGH03] Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. Extending arbitrary solvers with constraint handling rules. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 79–90. ACM Press, 2003.

[DSGH04] Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of constraint handling rules. In *20th International Conference on Logic Programming (ICLP'04)*, Saint-Malo, France, September 2004.

[Frü98] T. Frühwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Special Issue on Constraint Logic Programming*, volume 37, October 1998.

[Int03] Intelligent Systems Laboratory. *SICStus Prolog User's Manual*. PO Box 1263, SE-164 29 Kista, Sweden, October 2003.

[MSJ94] K. Marriott, H. Søndergaard, and N.D. Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

[Sch04] Tom Schrijvers. CHR benchmarks and programs, October 2004. Available at http://www.cs.kuleuven.ac.be/~toms/Research/CHR/.

[SD04a] T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, number 2004-01, 2004. ISSN 0939-5091.

[SD04b] Tom Schrijvers and Bart Demoen. Antimonotony-based delay avoidance for CHR. Report CW 385, K.U.Leuven, Department of Computer Science, jul 2004.

A The Refined Operational Semantics of CHRs

In this section we present the refined operational semantics of CHRs. This version differs from the original version in [DSGH04] in several ways, most notably in the way the **Solve** transition is defined. The original definition of **Solve** defined set S_1 to be all non-ground CHR constraints currently in the store. Although this definition is simple, it did not accurately describe what many systems actually implement in practice. For example, most Prolog implementations of CHRs only wakeup the constraints whose variables are directly affected by addition of c into the builtin store. It is possible to show (on some somewhat contrived examples) that this violates the original definition of the refined semantics.

We have modified the definition of **Solve** to capture a wider range of implementations. The new version of the refined operational semantics of CHRs is defined as follows:

Definition 2 (Refined Operational Semantics).
1. Solve

$$\langle [c|A], S, B, T \rangle_n \rightarrowtail \langle S_1 +\!\!+ A, S, c \wedge B, T \rangle_n$$

where c is a builtin constraint 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

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

in P and a substitution θ such that

$$\begin{cases} cons(H_1) = \theta(H_1') \\ cons(H_2) = \theta(H_2') \\ \mathcal{D} \models \neg(B \rightarrow \exists_r(\theta \wedge g)) \wedge (B \wedge c \rightarrow \exists_r(\theta \wedge g)) \end{cases}$$

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.

2. Activate

$$\langle [c|A], S, B, T\rangle_n \rightarrowtail \langle [c\#n : 1|A], \{c\#n\} \uplus S, B, T\rangle_{(n+1)}$$

where c is a CHR constraint (which has never been active).

3. Reactivate

$$\langle [c\#i|A], S, B, T\rangle_n \rightarrowtail \langle [c\#i : 1|A], S, B, T\rangle_n$$

where c is a CHR constraint (re-added to A by **Solve** but not yet active).

4. Drop

$$\langle [c\#i : j|A], S, B, T\rangle_n \rightarrowtail \langle A, S, B, T\rangle_n$$

where $c\#i : j$ is an occurrenced active constraint and there is no such occurrence j in P (all existing ones have already been tried thanks to transition 7).

5. Simplify

$$\begin{aligned} \langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail \\ \langle C +\!\!+ A, H_1 \uplus S, \theta \wedge B, T'\rangle_n \end{aligned}$$

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

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

and the matching substitution θ is such that

$$\begin{cases} c = \theta(d_j) \\ cons(H_1) = \theta(H_1') \\ cons(H_2) = \theta(H_2') \\ cons(H_3) = \theta(H_3') \\ \mathcal{D} \models B \rightarrow \exists_r(\theta \wedge g) \\ id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r] \notin T \end{cases}$$

In the result $T' = T \cup \{id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r]\}$.

6. Propagate

$$\begin{aligned} \langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail \\ \langle C +\!\!+ [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus S, \theta \wedge B, T'\rangle_n \end{aligned}$$

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

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

and the matching substitution θ is such that

$$\begin{cases} c = \theta(d_j) \\ cons(H_1) = \theta(H'_1) \\ cons(H_2) = \theta(H'_2) \\ cons(H_3) = \theta(H'_3) \\ \mathcal{D} \models B \rightarrow \exists_r(\theta \wedge g) \\ id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r] \notin T \end{cases}$$

In the result $T' = T \cup \{id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r]\}$.

The role of the propagation histories T and T' is exactly the same as with the theoretical operational semantics, ω_t.

7. Default

$$\langle [c\#i : j|A], S, B, T\rangle_n \rightarrowtail \langle [c\#i : j+1|A], S, B, T\rangle_n$$

if the current state cannot fire any other transition.

B Equivalence of Operational Semantics

In this section we present a proof of equivalence between the call-based ω_c and refined ω_r operational semantics. This involves showing that a derivation under the call-based semantics can be mapped to an equivalent derivation under the refined semantics, and vice versa.

B.1 From call-based to refined

Definition 3 (Sub-Computations). *We say D has* sub-computation D' *if either $D' = D$ or some transition in D is required to compute a derivation D'', and D' is a sub-computation of D''. We say D' is a* trivial *sub-computation of D if $D = D'$.*

Definition 4 (Refined). *We say a ω_c derivation*

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

is refined *if for all A' the following are ω_r derivations:*

$$\langle G +\!\!+ A', S, B, T\rangle_n \rightarrowtail^* \langle H +\!\!+ A', S', B', T'\rangle_{n'}$$

The usual definition of $+\!\!+$ (sequence concatenation) applies if G (or H) is a sequence, otherwise we use $a +\!\!+ A = [a|A]$. Note that if $G = \square$, then we consider $G +\!\!+ A = A$.

Lemma 1. *All ω_c derivations*

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

where all non-trivial sub-computations are refined, are refined.

Proof. By induction.

Base Case: Derivations containing no ω_c transitions, thus $\langle G, A, S, B, T\rangle_n = \langle H, A, S', B', T'\rangle_{n'}$ so it immediately follows that $\langle G +\!\!+ A', S, B, T\rangle_n = \langle H +\!\!+ A', S', B', T'\rangle_{n'}$.

Induction Step: Suppose all ω_c derivations D_i consisting of i transitions (and where all non-trivial sub-computations are refined), are refined. We show that the same is true for similar derivations of $i+1$ transitions.

Let $\sigma_i^c = \langle G_i, A, S_i, B_i, T_i\rangle_{n_i}$ be the last state in D_i. Let D_i^r be the corresponding ω_r derivation that exists because D_i is refined, and let $\sigma_i^r = \langle G_i +\!\!+ A', S_i, B_i, T_i\rangle_{n_i}$ be the last state in D_i^r. We consider all possible transitions between σ_i^c and σ_{i+1}^c in constructing a D_{i+1}, then show how to construct an ω_r derivation D_{i+1}^r which satisfies the definition of *refined*.

1. **Solve**: Then $G_i = c$ where c is a builtin constraint. Hence $\sigma_{i+1}^c = \langle \Box, A, S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}}$, where S_{i+1}, B_{i+1}, T_{i+1} and n_{i+1} are given by the subcomputation
$$\langle S_1, A, S_i, c \wedge B_i, T_i\rangle_{n_i} \rightarrowtail^* \langle \Box, A, S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}} \quad (1)$$
if $\mathcal{D} \models \bar{\exists}_{\emptyset} c \wedge B$, otherwise $S_{i+1} = S_i$, $B_{i+1} = c \wedge B_i$, $T_{i+1} = T_i$ and $n_{i+1} = n_i$ if $\mathcal{D} \models \neg(\bar{\exists}_{\emptyset} c \wedge B)$. By assumption, subcomputation (7) is refined, hence
$$\langle S_1 +\!\!+ A', S_i, c \wedge B_i, T_i\rangle_{n_i} \rightarrowtail^* \langle A', S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}} \quad (2)$$
Notice the last state is in the appropriate form for σ_{i+1}^r.
Let σ^r be the result of applying ω_r **Solve** to σ_i^r, then $\sigma^r = \langle S_1 +\!\!+ A', S_i, B_i, T_i\rangle_{n_i}$. Furthermore,
$$D_{i+1}^r = D_i^r \rightarrowtail_{solve} \sigma_r \rightarrowtail^* \sigma_{i+1}^r$$
by derivation (2). Hence D_{i+1} is also refined.
2. **Activate**: Then $G_i = c$ where c is a (non-numbered) CHR constraint. Hence $\sigma_{i+1}^c = \langle c\#n_i : 1, A, \{c\#n\} \uplus S_i, B_i, T_i\rangle_{(n_i+1)}$ and $\sigma_{i+1}^r = \langle [c\#n_i : 1|A'], \{c\#n\} \uplus S_i, B_i, T_i\rangle_{(n_i+1)}$ after applying ω_r **Activate** to σ_i^r. Hence D_{i+1} is also refined.
3. **Reactivate**: Then $G_i = c\#j$ a numbered CHR constraint. Hence $\sigma_{i+1}^c = \langle c\#j : 1, A, S_i, B_i, T_i\rangle_{n_i}$ and $\sigma_{i+1}^r = \langle [c\#j : 1|A'], S_i, B_i, T_i\rangle_{n_i}$ after applying ω_r **Reactivate** to σ_i^r. Hence D_{i+1} is also refined.
4. **Drop**: Then $G_i = c\#j : k$ where there is no such occurrence k of predicate c in P. Hence $\sigma_{i+1}^c = \langle \Box, A, S_i, B_i, T_i\rangle_{n_i}$ and $\sigma_{i+1}^r = \langle A', S_i, B_i, T_i\rangle_{n_i}$ after applying ω_r **Drop** to σ_i^r. Hence D_{i+1} is also refined.
5. **Simplify**: Then $G_i = c\#j : k$ and the conditions for ω_c hold. Assuming we can fire the rule, we have that $\sigma_{i+1}^c = \langle \Box, A, S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}}$ where the appropriate fields are given by the following subcomputation.
$$\begin{array}{c}\langle C, [c\#j : k|A], S_i - H_2 - H_3 - \{c\#j\}, \theta \wedge B_i, T_i'\rangle_{n_i} \\ \rightarrowtail^* \langle \Box, [c\#j : k|A], S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}}\end{array} \quad (3)$$

Where C, H_2, H_3, θ and T_{i+1} are defined by ω_c **Simplify** (see Section 3.2). By assumption, subcomputation (3) is refined, hence

$$\begin{aligned}\langle C +\!\!+ A', S_i - H_2 - H_3 - \{c\#j\}, \theta \wedge B_i, T_i' \rangle_{n_i} \rightarrowtail^* \\ \langle A', S_{i+1}, B_{i+1}, T_{i+1} \rangle_{n_{i+1}}\end{aligned} \tag{4}$$

Notice the last state is in the appropriate form for σ^r_{i+1}.
Let σ^r be the result of applying ω_r **Simplify** to σ^r_i, then $\sigma^r = \langle C +\!\!+ A', S_i - H_2 - H_3 - \{c\#j\}, \theta \wedge B_i, T_i' \rangle_{n_i}$. Furthermore,

$$D^r_{i+1} = D^r_i \rightarrowtail_{simplify} \sigma_r \rightarrowtail^* \sigma^r_{i+1}$$

by derivation (6). Hence D_{i+1} is also refined.
The other case is when the rule is not applicable. Then $\sigma^c_{i+1} = \langle c\#j : k+1, A, S_i, B_i, T_i \rangle_{n_i}$ and $\sigma^r_{i+1} = \langle [c\#j : k+1 | A'], S_i, B_i, T_i \rangle_{n_i}$ after applying ω_r **Default**. Hence D_{i+1} is also refined.

6. **Propagate**:
   Then $G_i = c\#j : k$ and the conditions for ω_c hold. We have that $\sigma^c_{i+1} = \langle G, A, S_{i+1}, B_{i+1}, T_{i+1} \rangle_{n_{i+1}}$ where the appropriate fields are given by the following series of subcomputations.

$$\begin{aligned}\langle C, [c\#j : k | A], S_{l-1} - H_{3l}, \theta \wedge B_{l-1}, T_{l-1} \rangle_{n_{l-1}} \rightarrowtail^* \\ \langle \Box, [c\#j : k | A], S_l, B_l, T_l \rangle_{n_l}\end{aligned} \tag{5}$$

   Where all of the appropriate fields are defined by ω_c **Propagate** (see Section 3.2). By assumption, each subcomputation (3) is refined, hence

$$\begin{aligned}\langle C +\!\!+ [c\#j : k | A'], S_{l-1} - H_{3l}, \theta \wedge B_{l-1}, T_{l-1} \rangle_{n_{l-1}} \rightarrowtail^* \\ \langle [c\#j : k | A'], S_l, B_l, T_l \rangle_{n_l}\end{aligned} \tag{6}$$

   Suppose that there are k sub-derivations, then we can construct ω_r derivation D^r_{i+1} as follows

$$D^r_{i+1} = D^r_i (D_l)^k \rightarrowtail_{default} \sigma^r_{i+1}$$

   where each sub-derivation D_l is constructed by applying **Propagate** (whose applicability is easily verified) to the last state in D_{l-1} (or σ^r_i if $l = 1$), and then the proceeding derivation defined by (6). Hence D_{i+1} is also refined.

7. **Goal**:
   Then $G_i = [c|C]$ is a sequence of constraints. Hence $\sigma^c_{i+1} = \langle C, A, S_{i+1}, B_{i+1}, T_{i+1} \rangle_{n_{i+1}}$, where S_{i+1}, B_{i+1}, T_{i+1} and n_{i+1} are given by the subcomputation

$$\langle c, A, S_i, B_i, T_i \rangle_{n_i} \rightarrowtail^* \langle \Box, A, S_{i+1}, B_{i+1}, T_{i+1} \rangle_{n_{i+1}} \tag{7}$$

   By assumption, subcomputation (7) is refined, hence

$$\begin{aligned}\langle [c|C] +\!\!+ A', S_i, B_i, T_i \rangle_{n_i} \rightarrowtail^* \\ \langle C +\!\!+ A', S_{i+1}, B_{i+1}, T_{i+1} \rangle_{n_{i+1}}\end{aligned} \tag{8}$$

   Notice the last state is in the appropriate form for σ^r_{i+1}.
   Now $\sigma^r_i = \langle [c|C] +\!\!+ A', S_i, B_i, T_i \rangle_{n_i}$, so we can now construct D^r_{i+1} as follows

$$D^r_{i+1} = D^r_i \rightarrowtail^* \sigma^r_{i+1}$$

   by derivation (8). Hence D_{i+1} is also refined.

Therefore ω_c derivations D where all non-trivial sub-computations are refined, are refined. □

The next Lemma is the same as the previous one except now some assumptions are removed.

Lemma 2. *All ω_c derivations D, of the form*

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

are refined.

Proof. By induction.

Base Case: D_0 has no non-trivial sub-computations, hence D_0 has no non-refined sub-computations, then by Lemma 1 all possible D_0 are refined.

Induction Step: Suppose that all D_i where the maximum depth of nested sub-computations of D_i is less than or equal to i, are refined. Then we show that similarly defined D_{i+1}, which only has (at least one) D_i sub-computations, is also refined. This also immediately follows from Lemma 1, since all sub-computations are refined.

Therefore all ω_c derivations D, of the form

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

, are refined. □

B.2 From refined to call-based

Definition 5 (Sub-computation). *Let $\sigma = \langle C +\!\!+ A, S, B, T\rangle_n$ be the result after applying a **Solve**, **Simplfy** or **Propagate** transition under ω_r. We interpret $C = S_1$ (see Definition 2) for the case of **Solve**, or C is the (renamed) body of the rule otherwise. We define a* sub-computation *to be the derivation*

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

where the final state is the only state in the derivation of that form.

Definition 6 (Call-based). *We say a ω_r derivation*

$$\langle G +\!\!+ A, S, B, T\rangle_n \rightarrowtail^* \langle A, S', B', T'\rangle_{n'}$$

is call-based *if for all A' the following are ω_c derivations*

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

The usual definition of $+\!\!+$ (sequence concatenation) applies if G is a sequence, otherwise we use $a +\!\!+ A = [a|A]$. Note that if $G = \square$, then we consider $G +\!\!+ A = A$.

Lemma 3. *All derivations of the form* $\langle [C|A], S, B, T\rangle_n \rightarrowtail^* \langle [C'|A], S', B', T'\rangle_{n'}$ *(where* C *and* C' *can be a builtin, CHR, numbered or active constraints) are call-based if*

1. *the derivation contains no* ***Solve****,* ***Drop*** *or* ***Simplify*** *transitions, except in subcompuations;*
2. *and all non-trivial sub-computations are call-based.*

Proof. By induction.

Base Case: Derivations containing no ω_r transitions, thus $\langle [C|A], S, B, T\rangle_n = \langle [C'|A], S', B', T'\rangle_{n'}$ so it immediately follows that $\langle C, A', S, B, T\rangle_n = \langle C', A', S', B', T'\rangle_{n'}$.

Induction Step: Suppose all ω_r derivations D_i (which satisfy our conditions), are call-based. Here i is the number of transitions in the corresponding ω_c derivation for D_i. We show that the same is true for similar derivations D_{i+1} transitions constructed from some D_i.

Let $\sigma_i^r = \langle [C_i|A], S_i, B_i, T_i\rangle_{n_i}$ be the last state in D_i. Let D_i^c be the corresponding ω_c derivation that exists because D_i is call-based, and let $\sigma_i^c = \langle C_i, A', S_i, B_i, T_i\rangle_{n_i}$ be the last state in D_i^c. We consider all possible transitions applicable to σ_i^r, then show how to construct a ω_r derivation D_{i+1} and a corresponding ω_c derivation D_{i+1}^c which satisfies the definition of *call-based*. Let $\sigma_i'^r$ be the result of applying an ω_r transition to σ_i^r.

1. **Activate**: Then $C_i = c$ where c is a (non-numbered) CHR constraint. Hence $\sigma_i'^r = \sigma_{i+1}^r = \langle [c\#n_i : 1|A], \{c\#n_i\} \uplus S_i, B_i, T_i\rangle_{(n_i+1)}$ and $\sigma_{i+1}^c = \langle c\#n_i : 1, A', \{c\#n_i\} \uplus S_i, B_i, T_i\rangle_{(n_i+1)}$ after applying ω_c **Activate**. Hence D_{i+1} is also call-based.
2. **Reactivate**: Then $C = c\#j$. Hence $\sigma_i'^r = \sigma_{i+1}^r = \langle [c\#j : 1|A], S_i, B_i, T_i\rangle_{n_i}$ and $\sigma_{i+1}^c = \langle c\#n_i : 1, A', S_i, B_i, T_i\rangle_{n_i}$ after applying ω_c **Reactivate**. Hence D_{i+1} is also call-based.
3. **Propagate**:
   Then $C_i = c\#j : k$ and the conditions for ω_r hold. We have that $\sigma_i'^r = \langle C +\!\!+ A, S_i - H_3, \theta \wedge B_i, T_i'\rangle_{n_i'}$ Where C, H_3, θ and T_i' are defined by ω_r **Propagate**. Consider the derivation

   $$D_{i+1}^r = D_i^r (D_l)^k \rightarrowtail_{default} \langle [c\#j : k+1|A], S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}} = \sigma_{i+1}^r$$

   where each sub-derivation D_l is of the form

   $$\sigma_l = \langle [c\#j : k|A], S_l, B_l, T_l\rangle_{n_l} \rightarrowtail_{propagate} \sigma_l'' \rightarrowtail^* \langle [c\#j : k|A], S_l', B_l', T_l'\rangle_{n_l'} = \sigma_l'$$

   where σ_l' is the first state in D_l of that form (excluding state σ_l). By assumption, sub-computations $\sigma_l'' \rightarrowtail^* \sigma_l'$ are call-based. We observe that

   $$\sigma_{i+1}^c = \langle c\#j : k+1, A', S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}}$$

   after applying ω_c **Propagate**. Hence D_{i+1} is also call-based.

4. **Default**: Then $C = c\#j : k$ and no other transitions is applicable to σ_i^r. Hence $\sigma_{i+1}^r = \langle [c\#j : k+1|A], S_i, B_i, T_i\rangle_{n_i}$ and $\sigma_{i+1}^c = \langle c\#j : k+1, A', S_i, B_i, T_i\rangle_{n_i}$ after applying ω_c **Simplify** or **Propagate** (depending on whether occurrence k deletes the active constraint or not). Hence D_{i+1} is also refined.

Therefore ω_c derivations D which satisfy our conditions are call-based. □

Lemma 4. *All derivations D of the form*

$$\langle [C|A], S, B, T\rangle_n \rightarrowtail^* \langle A, S', B', T'\rangle_{n'} = \sigma_f$$

(where C can be a builtin, CHR, numbered or active constraint) are call-based if

1. *σ_f is the only state in D of that form; and*
2. *and all non-trivial sub-computations are call-based.*

Proof. Direct proof. We can divide D as follows.

$$\sigma = \langle [C|A], S, B, T\rangle_n \rightarrowtail^* \langle [C'|A], S'', B'', T''\rangle_{n''} = \sigma' \rightarrowtail_{(solve \vee drop \vee simplify)} \rightarrowtail^* \sigma_f$$

The first part of the derivation $\sigma \rightarrowtail^* \sigma'$ satisfies the conditions for Lemma 3, hence this part of D is call based, thus

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

under the ω_c semantics.

We consider all possible transitions applicable to σ', and show the derivation must be call-based.

1. **Solve**: Then $C' = c$ where c is a builtin constraint. Hence

   $$\sigma' \rightarrowtail_{solve} \langle S_1 +\!\!+ A, S'', c \wedge B'', T''\rangle_{n''}$$

   By assumption, the subcomputation

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

   is call-based, thus

   $$\sigma'^c \rightarrowtail_{solve} \langle \Box, A', S', B', T'\rangle_{n'}$$

   Hence D is call-based.
2. **Drop**: Then $C' = c\#j : k$ where there is no such occurrence k of predicate c in P. Hence $\sigma' \rightarrowtail_{drop} \langle A, S'', B'', T''\rangle_{n''} = \sigma_f$ and $\sigma'^c \rightarrowtail \langle \Box, A', S'', B'', T''\rangle_{n''}$ after applying ω_c **Drop**. Hence D is call-based.
3. **Simplify**: Then $C' = c\#j : k$ and the conditions for ω_r hold. Hence

   $$\sigma' \rightarrowtail_{solve} \langle C +\!\!+ A, S'' - H_2 - H_3 - \{c\#j\}, \theta \wedge B'', T'''\rangle_{n'}$$

where C, H_2, H_3, θ and T''' are defined by ω_r **Simplify**. By assumption, the sub-computation

$$\langle C +\!\!+ A, S'' - H_2 - H_3 - \{c\#j\}, \theta \wedge B'', T'''\rangle_{n'} \rightarrowtail^* \langle A, S', B', T'\rangle_{n'}$$

is call-based, thus

$$\sigma'^c \rightarrowtail_{simplify} \langle \Box, A', S', B', T'\rangle_{n'}$$

Hence D is call-based.

Therefore, derivation D is call-based. □

Lemma 5. *All derivations D of the form*

$$\langle A +\!\!+ A', S, B, T\rangle_n \rightarrowtail^* \langle A', S', B', T'\rangle_{n'} = \sigma_f$$

(where A and A' contain only builtin or CHR constraints) are call-based if

1. *σ_f is the only state in D of that form;*
2. *and all non-trivial sub-computations are call-based.*

Proof. By induction.

Base case: Derivations containing no ω_r transitions, i.e. $A = []$, thus $\langle A +\!\!+ A', S, B, T\rangle_n = \langle A', S', B', T'\rangle_{n'}$ and so it immediately follows that $\langle A', A'', S, B, T\rangle_n = \langle A', A'', S', B', T'\rangle_{n'}$ which is a zero-length ω_c derivation of the appropriate form.

Induction step: Suppose that D_i is call-based for all derivations D_i of the following form

$$\langle A_i +\!\!+ A', S_i, B_i, T_i\rangle_{n_i} \rightarrowtail^* \langle A', S', B', T'\rangle_{n'}$$

where the length of A_i is i, and D_i satisfies the same conditions as D. We show the same is true for all similarly defined derivations D_{i+1}. Note $A_{i+1} = [C|A_i]$ for some C.

Let the first state in D_{i+1} be $\langle [C|A_i], S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}}$. Let D_i^c be the call-based derivation of the form

$$\langle A_i +\!\!+ A', A'', S_i, B_i, T_i\rangle_{n_i} \rightarrowtail^* \langle A', A'', S', B', T'\rangle_{n'}$$

that exists because D_i is call-based. By Lemma 4 the following is a call-based derivation

$$\langle C, A'', S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}} \rightarrowtail^* \langle \Box, A'', S_i, B_i, T_i\rangle_{n_i} \tag{9}$$

Consider the call-based state

$$\sigma_{i+1}^c = \langle [C|A_i] +\!\!+ A', A'', S_{i+1}, B_{i+1}, T_{i+1}\rangle_{n_{i+1}}$$

We can apply ω_c **Goal** to this state, which computes the sub-computation in (9) to arrive at $\sigma_i^c = \langle A_i +\!\!+ A', A'', S_i, B_i, T_i\rangle_{n_i}$. We have thus constructed a derivation D_{i+1}^c of the required form to show that D_{i+1} is call-based.

Therefore, all derivations D which satisfy our conditions are call-based. □

Lemma 6. *All ω_r derivations D, of the form*

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

where A' contain only builtin or CHR constraints, are call-based.

Proof. By induction.

Base Case: D_0 has no non-trivial sub-computations, hence D_0 has no non-call-based sub-computations, then by Lemma 5 all possible D_0 are call-based.

Induction Step: Suppose that all D_i where the maximum depth of nested sub-computations of D_i is less than or equal to i, are call-based. Then we show that similarly defined D_{i+1}, which only has (at least one) D_i sub-computations, is also refined. This also immediately follows from Lemma 5, since all sub-computations are refined.

Therefore all ω_r derivations D, of the form

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

are call-based. □

B.3 Main Result

Theorem 1 (Equivalence of Semantics). *For an initial goal G; $\langle G, [], true, \emptyset\rangle_1 \rightarrowtail^* \langle [], S, B, T\rangle_n$ under the refined semantics iff $\langle G, [], [], true, \emptyset\rangle_1 \rightarrowtail^* \langle [], [], S, B, T\rangle_n$ under the call-based semantics.*

Proof. Immediately follows from Lemma 2 and Lemma 6. □