The Refined Operational Semantics of Constraint Handling Rules[∗]

GREGORY J. DUCK, PETER J. STUCKEY
NICTA Victoria Laboratory
Department of Computer Science & Software Engineering, University of Melbourne, Australia
(*e-mail:* {gjd,pjs}@cs.mu.oz.au)

MARIA GARCIA DE LA BANDA
School of Computer Science & Software Engineering, Monash University, Australia
(*e-mail:* mbanda@csse.monash.edu.au)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Constraint Handling Rules (CHRs) are a high-level rule-based programming language commonly used to write constraint solvers. The theoretical operational semantics for CHRs is highly non-deterministic and relies on writing confluent programs to have a meaningful behaviour. Implementations of CHRs use an operational semantics which is considerably finer than the theoretical operational semantics, but is still non-deterministic (from the user's perspective). This paper formally defines this *refined* operational semantics and proves it implements the theoretical operational semantics. It also shows how to create a (partial) confluence checker capable of detecting programs which are confluent under this semantics, but not under the theoretical operational semantics. This supports the use of new idioms in CHR programs.

KEYWORDS: Constraint handling rules, operational semantics, confluence

1 Introduction

Constraint Handling Rules (CHRs) are a rule-based programming language originally designed to write constraint solvers in a high-level and declarative way. Increasingly CHRs are being used as a general purpose programming language, and several complex algorithms/programs have been implemented in CHRs, e.g. union-find algorithms (Schrijvers and Frühwirth 2006), ray tracing, Dijkstra's algorithm (Sneyers et al. 2006) and many more. The high-level nature of CHRs means that complex interactions may be expressed in just a few rules.

Operationally, CHRs exhaustively apply a set of rules to an initial set of constraints until a fixed point is reached. We refer to these operational semantics as *theoretical*, because they describe how CHRs are allowed to behave in theory. The

∗ A preliminary version of this paper appeared in the *International Conference on Logic Programming (ICLP2004)*, St Malo, France, 2004.

theoretical semantics is highly nondeterministic, and this is sometimes inconvenient from a programming language point of view. For example, the theoretical semantics do not specify which rule to apply if more than one possibility exists, and the final answer may depend on such a choice.

This paper defines the *refined* operational semantics for CHRs: a more deterministic operational semantics which has been implicitly described in (Holzbaur and Frühwirth 1999; Holzbaur and Frühwirth 2000), and is used by almost all modern implementations of CHRs we know of. Some choices are still left open in the refined operational semantics, however both the order in which constraints are executed and the order which rules are applied, is decided. Unsurprisingly, the decisions follow Prolog style and maximise efficiency of execution.

It is clear that CHR programmers take the refined operational semantics into account when programming. For example, some of the standard CHR examples are non-terminating under the theoretical operational semantics.

Example 1 *Consider the following simple program that calculates the greatest common divisor (*gcd*) between two integers using Euclid's algorithm:*

```
gcd1 @ gcd(0) <=> true.
gcd2 @ gcd(N) \ gcd(M) <=> M >= N | gcd(M-N).
```

Rule `gcd1` is a simplification rule. It states that a fact `gcd(0)` in the store can be replaced by true. Rule `gcd2` is a simpagation rule, it states that if there are two facts in the store `gcd(n)` and `gcd(m)` where $m \geq n$, we can replace the part after the slash `gcd(m)` by the right hand side `gcd(`$m-n$`)`.[1] *The idea of this program is to reduce an initial store of* `gcd(A), gcd(B)` *to a single constraint* `gcd(C)` *where* `C` *will be the* gcd *of* `A` *and* `B`*.*

This program, which appears on the CHR webpage (Schrijvers 2005), is nonterminating under the theoretical operational semantics. Consider the constraint store `gcd(3), gcd(0)`*. If the first rule fires, we are left with `gcd(3)` and the program terminates. If, instead, the second rule fires (which is perfectly possible in the theoretical semantics),* `gcd(3)` *will be replaced with* `gcd(3-0) = gcd(3)`*, thus essentially leaving the constraint store unchanged. If the second rule is applied indefinitely (assuming unfair rule application), we obtain an infinite loop.*

In the above example, trivial non-termination can be avoided by using a *fair* rule application (i.e. one in which every rule that could fire, eventually does). Indeed, the theoretical operational semantics given in (Frühwirth 1998) explicitly states that rule application should be fair. Interestingly, although the refined operational semantics is not fair (it uses rule ordering to determine rule application), its unfairness ensures termination in the `gcd` example above. Of course, it could also have worked against it, since swapping the order of the rules would lead to nontermination.

The refined operational semantics allows us to use more programming idioms, since we can now treat the constraint store as a queryable data structure.

[1] Unlike Prolog, we assume the function call "$m - n$" is evaluated as an integer subtraction.

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

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

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

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

Fig. 1. Naive union find algorithm impelemented in CHRs

Example 2 *Consider the naive union find algorithm implemented in CHRs (Schrijvers and Frühwirth 2006) and shown in Figure 1.*

We can classify the constraints in this program into two types: (1) those that represent the datastructure (the graph) and (2) those that are operations that either manipulate or query the datastructure. The constraints `root`/1 and `arrow`/2 belong to the former, `make`/1, `union`/2, `link`/2 and `find`/2 belong to the later.

This program is clearly non-confluent under the theoretical semantics, since the relative order of `find`/2 operation (which queries the arrows in the graph) and `link`/2 operation (which adds a new arrow to the graph) matters (Frühwirth 2005). Under the theoretical semantics the order of these operations is not necessarily preserved, however it is preserved under the refined semantics

The refined operational semantics also allows us to create more efficient programs and/or have a better idea regarding their time complexity.

Example 3 *Consider the following implementation of Fibonacci numbers, `fib(N,F)`, which holds if F is the N^{th} Fibonacci number:*

```
f1 @ fib(N,F) <=> 1 >= N | F = 1.
f2 @ fib(N,F0) \ fib(N,F) <=> N >= 2 | F = F0.
f3 @ fib(N,F) ==> N >= 2 | fib(N-2, F1), fib(N-1,F2), F = F1 + F2.
```

The program is confluent in the theoretical operational semantics which, as we will see later, means it is also confluent in the refined operational semantics. Under the refined operational semantics it has linear complexity, while swapping rules `f2` and `f3` leads to exponential complexity. Since in the theoretical operational semantics both versions are equivalent, worst case complexity is at best exponential.

We believe that CHRs under the refined operational semantics provide a powerful and elegant language suitable for general purpose computing. However, to make use of this language, authors need support to ensure their code is confluent within this context. In order to do this, we first provide a formal definition of the refined operational semantics of CHRs as implemented in logic programming systems. Essentially, these results ensure that if a program is confluent and terminating under the theoretical semantics, it is also confluent and terminating under the refined semantics. We then provide theoretical results linking the refined and theoretical operational semantics. We also discuss the consequences of the remaining choices left open in the refined semantics. Then, we provide a practical (partial) confluence test capable of detecting CHR programs which are confluent for the refined operational semantics, even though they are not confluent for the theoretical operational semantics. Finally, we study the accuracy of the confluence test over a collection of CHR programs.

The remainder of the article is organized as follows. In the next section we define the theoretical operational semantics ω_t for CHRs. Then in Section 3 we define the refined operational semantics ω_r. Then in Section 4 we define the relationship between the two operational semantics. We investigate how to define a confluence test for the refined operational semantics in Section 5. In Section 6 we give details about the practical construction of the confluence checker, and show the results on 4 benchmark programs. Finally in Section 7 we discuss related work, before concluding in Section 8.

2 The Theoretical Operational Semantics ω_t

We begin by defining constraints, rules and CHR programs. For our purposes, a *constraint* is simply defined as an atom $p(t_1, ..., t_n)$ where p is some predicate symbol of arity $n \geq 0$ and $(t_1, ..., t_n)$ is an n-tuple of terms. A *term* is defined as either a variable X, or as $f(t_1, ..., t_n)$ where f is a function symbol of arity n and $t_1, ..., t_n$ are terms. Let $vars(A)$ return the variables occurring in any syntactic object A. We use $\exists_A F$ to denote the formula $\exists X_1 \cdots \exists X_n F$ where $\{X_1, \dots X_n\} = vars(A)$. Similarly, we use $\bar{\exists}_A F$ to denote the formula $\exists X_1 \cdots \exists X_n F$ where $\{X_1, \dots X_n\} = vars(F) - vars(A)$. We define $\forall_A F$ and $\bar{\forall}_A F$ similarly.

We use $[H|T]$ to denote the first H and remaining elements T of a sequence, $++$ for sequence concatenation, $[]$ for empty sequences, and $\uplus$ for multiset union. We shall sometimes treat multisets as sequences, in which case we nondeterministically choose an order for the objects in the multiset. We use the notation $p(s_1, \dots, s_n) = p(t_1, \dots, t_n)$ as shorthand for the constraint $s_1 = t_1 \wedge \cdots \wedge s_n = t_n$, and similarly $S = T$ where S and T are equal length sequences $S \equiv s_1 \ \cdots \ s_n$ and $T = t_1 \ \cdots \ t_n$ as shorthand for $s_1 = t_1 \wedge \cdots \wedge s_n = t_n$.

Constraints can be divided into either *CHR* constraints or *built-in* constraints in some constraint domain $\mathcal{D}$. Decisions about rule matchings will rely on the underlying solver proving that the current constraint store for the underlying solver entails a *guard* (a conjunction of built-in constraints). We will assume the solver supports (at least) equality.

There are three types of rules: simplification, propagation and simpagation, which have the respective syntax:

$$
\begin{array}{rll}
r \ @ \ H & \Longleftrightarrow g \mid B & \qquad\qquad \textit{Simplification} \\
r \ @ \ H & \Longrightarrow g \mid B & \qquad\qquad \textit{Propagation} \\
r \ @ \ H_1 \setminus H_2 & \Longleftrightarrow g \mid B & \qquad\qquad \textit{Simpagation}
\end{array}
$$

where r is the rule name, H, H_1 and H_2 are non-empty sequences of CHR constraints, g is a sequence of built-in constraints, and B is a sequence of constraints.

For simplicity, we consider both simplification and propagation rules as special cases of generalised simpagation rules, where $H_2 = []$ is a propagation rule and $H_1 = []$ is a simplification rule. At least one of H_1 and H_2 must be non-empty. Finally, a CHR program P is a sequence of rules.

Given a CHR program P, we will be interested in numbering the occurrences of each CHR constraint predicate p appearing in the head of the rule. We number the occurrences following the top-down rule order and right-to-left constraint order. The latter is aimed at ordering first the constraints after the backslash (\) and then those before it, since this is more efficient in general.

Example 4 *The following shows the* `gcd` *CHR program of Example 1, written using simpagation rules and all occurrences numbered:*

```
gcd1 @ []        \ gcd(0)₁ <=> true   | true.
gcd2 @ gcd(N)₃  \ gcd(M)₂ <=> M ≥ N | gcd(M-N).
```

2.1 The ω_t Semantics

Several versions of the theoretical operational semantics have already appeared in the literature, e.g. (Abdennadher 1997; Frühwirth 1998), essentially as a multiset rewriting semantics. This section presents our variation,[2] which subsumes previous versions, and is close enough to our refined operational semantics to make proofs simple.

Firstly we define *numbered* constraints.

Definition 1 (Numbered Constraints) *A* numbered constraint *is a constraint c paired with an integer i. We write $c\#i$ to indicate a numbered constraint.*

Sometimes we refer to i as the *identifier* (or simply ID) of the numbered constraint. This numbering serves to differentiate among copies of the same constraint.

Now we define an *execution state*, as follows.

Definition 2 (Execution State) *An* execution state *is a tuple of the form $\langle G, S, B, T\rangle_n^{\mathcal{V}}$ where G is a multiset (repeats are allowed) of constraints, S is a set of numbered constraints, B is a conjunction of built-in constraints, T is a set of sequences of integers, $\mathcal{V}$ is the set of variables and n is an integer. Throughout this paper we use symbol 'σ' to represent an execution state.*

We call G the *goal*, which contains all constraints to be executed. The CHR constraint *store* S is the set[3] of *numbered* CHR constraints that can be matched with rules in the program P. For convenience we introduce functions $chr(c\#i) = c$ and

[2] A brief comparison between this and previous formalisations of the semantics can be found later in Section 7.
[3] Sometimes we treat the store as a multiset.

$id(c\#i) = i$, and extend them to sequences and sets of numbered CHR constraints in the obvious manner.

The *built-in constraint store* B contains any built-in constraint that has been passed to the built-in solver. Since we will usually have no information about the internal representation of B, we treat it as a conjunction of constraints. 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 (which may be represented as a unique integer, but typically we just use the name of the rule itself). This is necessary to prevent trivial nontermination 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. The set $\mathcal{V}$ contains all variables that appeared in the initial goal. Throughout this paper we will usually omit $\mathcal{V}$ unless we require it to be explicitly shown. Finally, the counter n represents the next free integer which can be used to number a CHR constraint.

We define an *initial state* as follows.

Definition 3 (Initial State) *Given a goal G, which is a multiset of constraints, the* initial state *with respect to G is $\langle G, \emptyset, true, \emptyset\rangle_1^{vars(G)}$.*

The theoretical operational semantics ω_t is based on the following three transitions which map execution states to execution states:

Definition 4 (Theoretical Operational Semantics)
1. Solve

$$\langle \{c\} \uplus G, S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle G, S, c \wedge B, T\rangle_n^{\mathcal{V}}$$

where c is a built-in constraint.
2. Introduce

$$\langle \{c\} \uplus G, S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle G, \{c\#n\} \uplus S, B, T\rangle_{(n+1)}^{\mathcal{V}}$$

where c is a CHR constraint.
3. Apply

$$\langle G, H_1 \uplus H_2 \uplus S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle C \uplus G, H_1 \uplus S, B \wedge \theta, T'\rangle_n^{\mathcal{V}}$$

where there exists a (renamed apart) rule in P of the form

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

and $\theta \equiv chr(H_1) = H_1' \wedge chr(H_2) = H_2'$ such that

$$\begin{cases} \mathcal{D} \models B \rightarrow \exists_r(\theta \wedge g) \\ id(H_1) +\!\!+ \; id(H_2) +\!\!+ \; [r] \notin T \end{cases}$$

In the result $T' = T \cup \{id(H_1) +\!\!+ \; id(H_2) +\!\!+ \; [r]\}$.[4]

[4] Note in practice we only need to keep track of tuples where H_2 is empty, since otherwise these CHR constraints are being deleted and the firing can not reoccur.

$$\begin{array}{llll}
 & & \langle\{\mathtt{gcd}(6),\mathtt{gcd}(9)\},\emptyset\rangle_1 & (1)\\
 & \rightarrowtail_{introduce} & \langle\{\mathtt{gcd}(9)\},\{\mathtt{gcd}(6)\#1\}\rangle_2 & (2)\\
 & \rightarrowtail_{introduce} & \langle\emptyset,\{\mathtt{gcd}(6)\#1,\mathtt{gcd}(9)\#2\}\rangle_3 & (3)\\
(\mathtt{gcd2}\ \ N=6\wedge M=9) & \rightarrowtail_{apply} & \langle\{\mathtt{gcd}(3)\},\{\mathtt{gcd}(6)\#1\}\rangle_3 & (4)\\
 & \rightarrowtail_{introduce} & \langle\emptyset,\{\mathtt{gcd}(6)\#1,\mathtt{gcd}(3)\#3\}\rangle_4 & (5)\\
(\mathtt{gcd2}\ \ N=3\wedge M=6) & \rightarrowtail_{apply} & \langle\{\mathtt{gcd}(3)\},\{\mathtt{gcd}(3)\#3\}\rangle_4 & (6)\\
 & \rightarrowtail_{introduce} & \langle\emptyset,\{\mathtt{gcd}(3)\#3,\mathtt{gcd}(3)\#4\}\rangle_5 & (7)\\
(\mathtt{gcd2}\ \ N=3\wedge M=3) & \rightarrowtail_{apply} & \langle\{\mathtt{gcd}(0)\},\{\mathtt{gcd}(3)\#3\}\rangle_5 & (8)\\
 & \rightarrowtail_{introduce} & \langle\emptyset,\{\mathtt{gcd}(3)\#3,\mathtt{gcd}(0)\#5\}\rangle_6 & (9)\\
(\mathtt{gcd1}) & \rightarrowtail_{apply} & \langle\emptyset,\{\mathtt{gcd}(3)\#3\}\rangle_6 & (10)
\end{array}$$

Fig. 2. ω_t derivation for `gcd`.

The **Solve** transition adds a new built-in constraint from goal G to the built-in store B. The **Introduce** transition adds a new numbered CHR constraint to the CHR store S. The **Apply** transition chooses a rule from the program such that matching constraints exist in the CHR store S, and the guard is entailed by the built-in store B, and fires it. For readability, we usually treat θ as a substitution and apply it to all relevant fields in the execution state, i.e. G, S and B. This does not affect the meaning of the execution state, or its transition applicability, but it helps remove the build-up of too many variables and constraints.

The theoretical operational semantics states that given a goal G, we nondeterministically apply the transitions from Definition 4 until a *final state* is reached. We define a final state as follows.

Definition 5 (Final States) *An execution state $\sigma = \langle G, S, B, T\rangle_n$ is a* final state *if either no transition defined in Definition 4 is applicable to σ, or $\mathcal{D} \models \neg\bar{\exists}_\emptyset B$ holds (often we simply use $false$ to represent such a state).*

The sequence of execution states generated by continuously applying transition steps is called a *derivation*, which is formally defined as follows.

Definition 6 (Derivation) *A derivation D is a non-empty (but possibly infinite) sequence of execution states $D = [\sigma_0, \sigma_1, \sigma_2, ...]$ such that σ_{i+1} is the result of applying a transition from Definition 4 to execution state σ_i for all consecutive states σ_i and σ_{i+1} in D.*

Usually we write $\sigma_0 \rightarrowtail \sigma_1 \rightarrowtail \sigma_2 \rightarrowtail ...$ instead of $[\sigma_0, \sigma_1, \sigma_2, ...]$ to denote a derivation, and the length of the derivation is the length of the sequence less one. Sometimes we use the notation $D = D_0 +\!\!+ D_1$ to represent a partition of derivation D.

Example 5 *Figure 2 is a (terminating) derivation under ω_t for the query* `gcd(6), gcd(9)` *executed on the* `gcd` *program in Example 4. For brevity, B, T and $\mathcal{V}$ have been removed from each tuple, and we apply the substitutions θ throughout. No more transitions on state $\langle\emptyset,\{\mathtt{gcd}(3)\#3\}\rangle_6$ are possible, so this is the final state.*

3 The Refined Operational Semantics ω_r

The *refined* operational semantics establishes an order for the constraints in G. As a result, we are no longer free to pick any constraint from G to either **Solve** or **Introduce** into the store. It also treats CHR constraints as procedure calls: each newly added CHR 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 might be executed and, when they finish, the execution returns to finding rules for the current constraint. Not surprisingly, this approach is used exactly because it corresponds closely to that of the language we compile to.

Formally, the execution state of the refined semantics is the tuple

$$\langle A, S, B, T\rangle_n^{\mathcal{V}}$$

where S, B, T, $\mathcal{V}$ and n, representing the CHR store, built-in store, propagation history, initial variables and next free identity number respectively, are exactly as with Definition 2. The *execution stack* A is a sequence of constraints, numbered CHR constraints and *active* CHR constraints, with a strict ordering in which only the top-most constraint is considered for execution.[5] We now define *active constraints*, which represent a specific call to a CHR constraint.

Definition 7 (Active Constraints) *An* active *constraint $c\#i : j$ is a numbered CHR constraint $c\#i$ associated with an integer j which represents the* occurrence *of predicate c in P the constraint $c\#i$ is allowed to match with.*

Unlike in the theoretical operational semantics, a numbered constraint may simultaneously appear in both the execution stack A and the store S.

Given initial goal G, the initial state is as before, i.e. of the form

$$\langle G, \emptyset, true, \emptyset\rangle_1^{vars(G)}$$

except this time G is an ordered sequence, rather than a constraint multiset. Just as with the theoretical operational semantics, execution proceeds by exhaustively applying transitions to the initial execution state until the built-in solver state is unsatisfiable or no transitions are applicable.

The refined operational semantics treats CHR constraints with only *fixed variables* as a special case. We formally define *fixed variable* as follows.

Definition 8 (Fixed) *Let B be a built-in store, then $v \in fixed(B)$ if*

$$\mathcal{D} \models \forall v \forall \rho(v)(\bar{\exists}_v(B) \wedge \bar{\exists}_{\rho(v)}\rho(B) \rightarrow v = \rho(v))$$

for arbitrary renaming ρ.

Informally, a variable which can only take one value to satisfy B is fixed. We say a constraint c is fixed if $vars(c) \subseteq fixed(B)$.

[5] The execution stack is analogous to a call-stack in other programming languages.

When a built-in constraint is added to the built-in store, the refined semantics *wakes up* a subset of the CHR store to be reconsidered for execution. The exact subset is left open, however it must satisfy the conditions of a *wakeup policy*, which is defined as follows.

Definition 9 (Wakeup Policy) *Let S be a CHR store, c a built-in constraint and B a built-in store, then a* wakeup policy *is a function $wakeup\_policy(S, c, B) = S_1$ where S_1 is a finite multiset such that for all $s \in S_1$ we have that $s \in S$, and S_1 satisfies the following further conditions:*

1. lower bound*: For all $M = H_1 +\!\!+ H_2 \subseteq S$ such that there exists a (renamed apart) rule*
$$r \ @ \ H'_1 \setminus H'_2 \Longleftrightarrow g \mid C$$
and $\theta \equiv chr(H_1) = H'_1 \wedge chr(H_2) = H'_2$ such that
$$\begin{cases} \mathcal{D} \not\models (B \rightarrow \exists_r(\theta \wedge g)) \\ \mathcal{D} \models (B \wedge c \rightarrow \exists_r(\theta \wedge g)) \end{cases}$$
then $M \cap S_1 \neq \emptyset$
2. upper bound*: If $m \in S_1$ then $vars(m) \not\subseteq fixed(B)$.*

Each implementation of the refined semantics provides its own wakeup policy. The *lower bound* ensures that S_1 is (at least) the minimum subset of the store that actually needs to be reconsidered thanks to the addition of c. The *upper bound* ensures that S_1 contains no fixed constraints. This will ensure that fixed constraints have a more deterministic behaviour, which is essential for confluence (see Section 5). Note that the definition of a wakeup policy also allows the set S_1 to contain multiple (redundant) copies of constraints in S.[6]

We can now define the refined operational semantics of CHRs.

Definition 10 (Refined Operational Semantics)
1. Solve
$$\langle [c|A], S, B, T \rangle_n^{\mathcal{V}} \rightarrowtail \langle wakeup\_policy(S, c, B) +\!\!+ A, S, c \wedge B, T \rangle_n^{\mathcal{V}}$$
where c is a built-in constraint.
2. Activate
$$\langle [c|A], S, B, T \rangle_n^{\mathcal{V}} \rightarrowtail \langle [c\#n : 1|A], \{c\#n\} \uplus S, B, T \rangle_{(n+1)}^{\mathcal{V}}$$
where c is a CHR constraint (which has never been active).
3. Reactivate
$$\langle [c\#i|A], S, B, T \rangle_n^{\mathcal{V}} \rightarrowtail \langle [c\#i : 1|A], S, B, T \rangle_n^{\mathcal{V}}$$

[6] This models the behaviour of a CHR constraint waking up more than once when a built-in constraint is added to the store, which may happen in practice.

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

4. Drop

$$\langle [c\#i : j|A], S, B, T\rangle_n^{\mathcal{V}} \rightarrowtail \langle A, S, B, T\rangle_n^{\mathcal{V}}$$

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

5. Simplify

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

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 $\theta \equiv chr(H_1) = H_1' \wedge chr(H_2) = H_2' \wedge c = d_j \wedge chr(H_3) = H_3'$ such that

$$\begin{cases} \mathcal{D} \models B \rightarrow \exists_r(\theta \wedge g) \\ id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r] \not\in T \end{cases}$$

In the result $T' = T \cup \{id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3) +\!\!+ [r]\}$.[7]

6. Propagate

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

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 $\theta \equiv chr(H_1) = H_1' \wedge c = d_j \wedge chr(H_2) = H_2' \wedge chr(H_3) = H_3'$ such that

$$\begin{cases} \mathcal{D} \models B \rightarrow \exists_r(\theta \wedge g) \\ id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r] \not\in T \end{cases}$$

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

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

if the current state cannot fire any other transition.

Some of the transitions for the refined operational semantics are analogous to transitions under the theoretical semantics. For example, the refined **Solve** transition corresponds to the theoretical **Solve** transition, as they both introduce a new built-in constraint into store B. Likewise, **Activate** corresponds to **Introduce** (add a new CHR constraint to the store), and **Simplify** and **Propagate**

[7] As with the theoretical semantics, it is not necessary to check the history if H_2 is not empty. We include the check anyway to simplify our proofs later in this paper.

correspond to **Apply** (fires a rule on some constraints in the store). This correspondence is not accidental, and later in this paper we formally define a mapping between the semantics.

The main difference between the refined and theoretical semantics is the presence (and behaviour) of *active* constraints. A rule r can only fire (via **Simplify** or **Propagate**) if the occurrence number of the current active constraint (on top of the execution stack) appears in the head of rule r. An active constraint is only allowed to match against the constraint in the head of r that shares the same occurrence number.

Example 6 *For example, an active constraint* $\mathtt{gcd}(3)\#4:2$ *(with occurrence number* 2*) is only allowed to match against* $\mathtt{gcd}(M)_2$ *(also with occurrence number* 2*) in the following rule.*

```
gcd2 @ gcd(N)₃ \ gcd(M)₂ <=> M ≥ N | gcd(M-N).
```

This is very different to the theoretical semantics, where a rule is free to fire on any subset of the current CHR store.

If the current active constraint cannot match against the associated rule, e.g. if all matchings have already been tried, then the active constraint “moves” to the next occurrence via the **Default** transition. This ensures that, assuming termination, all occurrences will eventually be tried. When there are no more occurrences to check, i.e. the occurrence number associated to an active constraint does not appear in P, then we can apply **Drop**. This pops off the current active constraint, but does not remove anything from the store.

Initially there are no active constraints in the goal, but we can turn a non-active constraint into an active constraint via the **Activate** transition. The **Activate** transition associates the first occurrence (defined as occurrence number 1) with the constraint, and adds a copy of the constraint to the CHR constraint store (just like **Introduce** under the theoretical semantics).

The **Solve** transition is also handled differently. After applying **Solve**, a subset of the CHR store (defined by the wakeup policy) is appended to the front of the execution stack. The intention is that these constraints will eventually become active again (via the **Reactivate** transition), and will reconsider all rules in the program P. These active constraints may fire against more rules than before, because the addition of the new built-in constraint c may mean the underlying solver can prove more guards hold than before.

Example 7 *Consider the following (canonical example) CHR program defining a less-than-or-equal-to relation* `leq`*.*

```
reflexivity  @ leq(X,X)₁ <=> true.
antisymmetry @ leq(X,Y)₃,  leq(Y,X)₂ <=> X = Y.
idempotence  @ leq(X,Y)₅ \ leq(X,Y)₄ <=> true.
transitivity @ leq(X,Y)₇,  leq(Y,Z)₆ ==> leq(X,Z).
```

$$\begin{array}{lll}
 & \langle[\mathtt{gcd}(6),\mathtt{gcd}(9)],\emptyset\rangle_1 & (1)\\
\rightarrowtail_{activate} & \langle[\mathtt{gcd}(6)\#1:1,\mathtt{gcd}(9)],\{\mathtt{gcd}(6)\#1\}\rangle_2 & (2)\\
\rightarrowtail^{\times 3}_{default} & \langle[\mathtt{gcd}(6)\#1:4,\mathtt{gcd}(9)],\{\mathtt{gcd}(6)\#1\}\rangle_2 & (2)\\
\rightarrowtail_{drop} & \langle[\mathtt{gcd}(9)],\{\mathtt{gcd}(6)\#1\}\rangle_2 & (2)\\
\rightarrowtail_{activate}\rightarrowtail_{default} & \langle[\mathtt{gcd}(9)\#2:2],\{\mathtt{gcd}(9)\#2,\mathtt{gcd}(6)\#1\}\rangle_3 & (3)\\
\rightarrowtail_{simplify} & \langle[\mathtt{gcd}(3)],\{\mathtt{gcd}(6)\#1\}\rangle_3 & (4)\\
\rightarrowtail_{activate}\rightarrowtail^{\times 2}_{default} & \langle[\mathtt{gcd}(3)\#3:3],\{\mathtt{gcd}(3)\#3,\mathtt{gcd}(6)\#1\}\rangle_3 & (5)\\
\rightarrowtail_{propagate} & \langle[\mathtt{gcd}(3),\mathtt{gcd}(3)\#3:3],\{\mathtt{gcd}(3)\#3\}\rangle_4 & (6)\\
\rightarrowtail_{activate}\rightarrowtail_{default} & \langle[\mathtt{gcd}(3)\#4:2,\mathtt{gcd}(3)\#3:3],\{\mathtt{gcd}(3)\#4,\mathtt{gcd}(3)\#3\}\rangle_5 & (7)\\
\rightarrowtail_{simplify} & \langle[\mathtt{gcd}(0),\mathtt{gcd}(3)\#3:3],\{\mathtt{gcd}(3)\#3\}\rangle_5 & (8)\\
\rightarrowtail_{activate} & \langle[\mathtt{gcd}(0)\#5:1,\mathtt{gcd}(3)\#3:3],\{\mathtt{gcd}(0)\#5,\mathtt{gcd}(3)\#3\}\rangle_6 & (9)\\
\rightarrowtail_{simplify} & \langle[\mathtt{gcd}(3)\#3:3],\{\mathtt{gcd}(3)\#3\}\rangle_6 & (10)\\
\rightarrowtail_{default}\rightarrowtail_{drop} & \langle[],\{\mathtt{gcd}(3)\#3\}\rangle_6 & (10)
\end{array}$$

Fig. 3. ω_r derivation for `gcd`.

Assume the built-in store B is empty (i.e. $B = true$), then a constraint $\mathtt{leq}(J, K)$ *where J and K are distinct variables cannot fire the first rule because $\mathcal{D} \models true \rightarrow \exists X((J = X \wedge K = X) \wedge true)$ does not hold. However, if a new constraint $J = K$ were to be added into the built-in store via a* ***Solve*** *transition, then $\mathcal{D} \models (J = K) \rightarrow \exists X(((J = X \wedge K = X) \wedge true)$ holds, thus the rule can now fire.*

Under the refined semantics, the constraint $\mathtt{leq}(J, K)$ *will be copied to the execution stack during the* ***Solve*** *transition. Therefore eventually (assuming termination), the constraint will be* ***Reactivate****d, and fire the rule.*

The refined operational semantics constrains the values a wakeup policy is allowed to return. At the very minimum, a wakeup policy must contain a CHR constraint from every new matching that is possible thanks to the addition of c into the built-in store (the *lower bound* condition). At most, a wakeup policy returns all non-ground constraints currently in the CHR store (the *upper bound* condition). The exact behaviour of the wakeup policy is left to the implementation.

We now present an example derivation under the refined operational semantics.

Example 8 *Figure 3 shows the derivation under ω_r semantics for the* `gcd` *program in Example 4 and the goal* `gcd(6),gcd(9)`*. For brevity B, T and $\mathcal{V}$ have been eliminated, and we apply the substitutions θ throughout.*

4 The relationship between ω_t and ω_r

Once both semantics are established, we can define an abstraction function α which maps execution states of ω_r to ω_t. Later, we use this abstraction function to prove correctness of the refined semantics ω_r with respect to the theoretical semantics ω_t.

Definition 11 (Correspondence of States) *The abstraction function α is defined as*

$$\alpha(\langle A, S, B, T\rangle_n^{\mathcal{V}}) = \langle no\_id(A), S, B, T\rangle_n^{\mathcal{V}}$$

where $no\_id(A) = \{c \mid c \in A$ is not of the form $c\#i$ or $c\#i : j\}$.

The abstraction function removes all numbered constraints from the execution stack, and turns the stack into an unordered multiset.

Example 9 *Consider the following ω_r state from the `gcd` example.*

$$\langle[\texttt{gcd}(0), \texttt{gcd}(3)\#3 : 3], \{\texttt{gcd}(3)\#3\}\rangle_5$$

After applying α we get

$$\langle\{\texttt{gcd}(0)\}, \{\texttt{gcd}(3)\#3\}\rangle_5$$

We have simply removed the active constraint `gcd(3)#3 : 3` *from the stack, as it was identified with number 3, and turned the stack (which is a sequence) into a multiset. The rest of the state is unaffected.*

We now extend α to map a derivation D under ω_r to the corresponding derivation $\alpha(D)$ under ω_t, by mapping each state appropriately and eliminating adjacent equivalent states.

Definition 12 (Correspondence of derivations) *Function α is extended to derivations in ω_r as follows*

$$\alpha(\sigma_1 \rightarrowtail D) = \begin{array}{ll} \alpha(D) & \textit{if } (D = \sigma_2 \rightarrowtail D' \textit{ or } D = \sigma_2) \textit{ and } \alpha(\sigma_1) = \alpha(\sigma_2) \\ \alpha(\sigma_1) \rightarrowtail \alpha(D) & \textit{otherwise} \end{array}$$

Note that this definition is just syntactic, and we do not know if the result of apply function α to a ω_r derivation gives a valid ω_t derivation. We rely on the following theorem to show this.

Theorem 1 (Correspondence) *For all ω_r derivations D, $\alpha(D)$ is a ω_t derivation.*

Proof
By induction. We use D_i to represent a derivation of length i, we also let σ_j be the j^{th} state in D_i, so $D_i = \sigma_0 \rightarrowtail \sigma_1 \rightarrowtail ... \rightarrowtail \sigma_i$. Derivations of length $i + 1$ must be constructed from derivations of length i.
Base case: Derivations of zero length. Then $\alpha(D_0) = \alpha(\sigma_0)$ is a ω_t derivation of zero length for all D_0.
Induction Step: Assume that for all derivations D_i of length i, $\alpha(D_i)$ is a ω_t derivation. Let D_{i+1} be a ω_r derivation of length $i + 1$ constructed from a derivation D_i by applying a ω_r transition to σ_i (the last state in D_i). We show that $\alpha(D_{i+1})$ is a ω_t derivation.

Let σ_{i+1} be the last state in D_{i+1}, we factor out two possible relationships between $\alpha(\sigma_i)$ and $\alpha(\sigma_{i+1})$ and show that these imply $\alpha(D_{i+1})$ is also a ω_t derivation.

- Case 1: $\alpha(\sigma_{i+1}) = \alpha(\sigma_i)$, then

$$\begin{array}{ll} \alpha(D_{i+1}) & = \\ \alpha(D_i \rightarrowtail \sigma_{i+1}) & = \\ \alpha(D_i) & \end{array}$$

Hence $\alpha(D_{i+1})$ is also a ω_t derivation; or

- Case 2: $\alpha(\sigma_i) \rightarrowtail_{\omega_t} \alpha(\sigma_{i+1})$ holds for some ω_t transition $\rightarrowtail_{\omega_t}$, then

$$\begin{array}{ll} \alpha(D_{i+1}) & = \\ \alpha(D_i \rightarrowtail_{\omega_r} \sigma_{i+1}) & = \\ \alpha(D_i) \rightarrowtail_{\omega_t} \alpha(\sigma_{i+1}) & \end{array}$$

This means $\alpha(D_{i+1})$ is constructed from ω_t derivation $\alpha(D_i)$ by applying a ω_t transition to the last state $\alpha(\sigma_i)$, hence $\alpha(\sigma_{i+1})$ is a ω_t derivation by definition.

It remains to be shown that if $\sigma_i \rightarrowtail \sigma_{i+1}$ under ω_r, then either Case 1 or Case 2 holds for $\alpha(\sigma_i)$ and $\alpha(\sigma_{i+1})$. For this we consider all possibilities for the ω_r transition from σ_i to σ_{i+1}.
CASE **Solve**: $\sigma_i \rightarrowtail \sigma_{i+1}$ is of the form

$$\langle [c|A], S, B, T\rangle_n \rightarrowtail_{solve} \langle \mathit{wakeup\_policy}(S, c, B) +\!\!+ A, S, c \wedge B, T\rangle_n$$

Where c is some built-in constraint. Then $\alpha(\sigma_i) = \langle \{c\} \uplus \mathit{no\_id}(A), S, B, T\rangle_n$. We can apply the ω_t version of **Solve** to $\alpha(\sigma_i)$.

$$\langle \{c\} \uplus \mathit{no\_id}(A), S, B, T\rangle_n \rightarrowtail_{solve} \langle \mathit{no\_id}(A), S, c \wedge B, T\rangle_n$$

Now $\langle \mathit{no\_id}(A), S, c \wedge B, T\rangle_n = \alpha(\sigma_{i+1})$, so $\alpha(\sigma_i) \rightarrowtail_{\omega_t} \alpha(\sigma_{i+1})$ where the transition is ω_t **Solve**. Hence Case 2 above is satisfied.
CASE **Activate**: $\sigma_i \rightarrowtail \sigma_{i+1}$ is of the form

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

Where c is some CHR constraint (not yet numbered). Then $\alpha(\sigma_i) = \langle \{c\} \uplus \mathit{no\_id}(A), S, B, T\rangle_n$. We can apply the ω_t **Introduce** to $\alpha(\sigma_i)$.

$$\langle \{c\} \uplus \mathit{no\_id}(A), S, B, T\rangle_n \rightarrowtail_{introduce} \langle \mathit{no\_id}(A), \{c\#n\} \uplus S, B, T\rangle_{(n+1)}$$

Now $\langle \mathit{no\_id}(A), \{c\#n\} \uplus S, B, T\rangle_{(n+1)} = \alpha(\sigma_{i+1})$, so $\alpha(\sigma_i) \rightarrowtail_{introduce} \alpha(\sigma_{i+1})$ where the ω_t transition is **Introduce**. Hence Case 2 above is satisfied.
CASE **Reactivate**: $\sigma_i \rightarrowtail \sigma_{i+1}$ is of the form

$$\langle [c\#i|A], S, B, T\rangle_n \rightarrowtail_{reactivate} \langle [c\#i : 1|A], S, B, T\rangle_n$$

Then $\alpha(\sigma_i) = \alpha(\sigma_{i+1}) = \langle \mathit{no\_id}(A), S, B, T\rangle_n$. Hence Case 1 above is satisfied.
CASE **Drop**: $\sigma_i \rightarrowtail \sigma_{i+1}$ is of the form

$$\langle [c\#i : j|A], S, B, T\rangle_n \rightarrowtail_{drop} \langle A, S, B, T\rangle_n$$

Then $\alpha(\sigma_i) = \alpha(\sigma_{i+1}) = \langle \mathit{no\_id}(A), S, B, T\rangle_n$. Hence Case 1 above is satisfied.
CASE **Simplify**: $\sigma_i \rightarrowtail \sigma_{i+1}$ is of the form

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

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 θ is $\theta \equiv chr(H_1) = H'_1 \wedge chr(H_2) = H'_2 \wedge c = d_j \wedge chr(H_3) = H'_3$, and the tuple $id(H_1) +\!\!+\ id(H_2) +\!\!+\ [i] +\!\!+\ id(H_3) +\!\!+\ [r] \not\in T$.

Then $\alpha(\sigma_i) = \langle no\_id(A), \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n$. We show that the ω_t **Apply** transition is applicable to $\alpha(\sigma_i)$, namely

- There exists a (renamed apart) rule in P of the form
$$r'' \ @\ H''_1 \setminus H''_2 \iff g'' \mid C''$$
This is satisfied by $r = r''$, $H''_1 = H'_1$, $H''_2 = (H'_2, d_j, H'_3)$, $g'' = g$ and $C'' = C$. In other words, the same rule as above.
- There exists a θ'' such that $\theta'' \equiv chr(H_1) = H''_1 \wedge chr(H_2, c, H_3) = H''_2$. This is satisfied by $\theta'' = \theta$ from above.
- The guard g'' is satisfied, i.e. $\mathcal{D} \models B \rightarrow \bar{\exists}_{r''}(\theta'' \wedge g'')$. This is satisfied because $\mathcal{D} \models B \rightarrow \bar{\exists}_r(\theta \wedge g)$ holds, and $r'' = r$, $\theta'' = \theta$ and $g'' = g$.
- The tuple $id(H_1) +\!\!+\ id(H_2) +\!\!+\ [i] +\!\!+\ id(H_3) +\!\!+\ [r] \not\in T$. This is directly satisfied from above.

Hence

$\langle no\_id(A), \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail_{apply} \langle C \uplus no\_id(A), H_1 \uplus S, \theta \wedge B, T'\rangle_n$

Now $\langle C \uplus no\_id(A), H_1 \uplus S, \theta \wedge B, T'\rangle_n = \alpha(\sigma_{i+1})$, so $\alpha(\sigma_i) \rightarrowtail_{apply} \alpha(\sigma_{i+1})$ where the ω_t transition is **Apply**. Hence Case 2 above is satisfied.

CASE **Propagate**: $\sigma_i \rightarrowtail \sigma_{i+1}$ is of the form

$$\langle [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail_{propagate}$$
$$\langle \theta(C) +\!\!+\ [c\#i : j|A], \{c\#i\} \uplus H_1 \uplus H_3 \uplus S, B, T'\rangle_n$$

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

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

and θ is $\theta \equiv chr(H_1) = H'_1 \wedge c = d_j \wedge chr(H_2) = H'_2 \wedge chr(H_3) = H'_3$, and $\mathcal{D} \models B \rightarrow \bar{\exists}_r(\theta \wedge g)$, and the tuple $id(H_1) +\!\!+\ [i] +\!\!+\ id(H_2) +\!\!+\ id(H_3) +\!\!+\ [r] \not\in T$.

Then $\alpha(\sigma_i) = \langle no\_id(A), \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n$. We show that the ω_t **Apply** transition is applicable to $\alpha(\sigma_i)$, namely

- There exists a (renamed apart) rule in P of the form
$$r'' \ @\ H''_1 \setminus H''_2 \iff g'' \mid C''$$
This is satisfied by $r = r''$, $H''_1 = (H'_1, d_j, H'_2)$, $H''_2 = H'_3$, $g'' = g$ and $C'' = C$. In other words, the same rule as above.
- There exists a θ'' such that $\theta'' \equiv chr(H_1, c, H_2) = H''_1 \wedge chr(H_3) = H''_2$. This is satisfied by $\theta'' = \theta$ from above.
- The guard g'' is satisfied, i.e. $\mathcal{D} \models B \rightarrow \bar{\exists}_{r''}(\theta'' \wedge g'')$. This is satisfied because $\mathcal{D} \models B \rightarrow \bar{\exists}_r(\theta \wedge g)$ holds, and $r'' = r$, $\theta'' = \theta$ and $g'' = g$.
- The tuple $id(H_1) +\!\!+\ [i] +\!\!+\ id(H_2) +\!\!+\ id(H_3) +\!\!+\ [r] \not\in T$. This is directly satisfied from above.

Hence

$$\begin{aligned}&\langle no\_id(A), \{c\#i\} \uplus H_1 \uplus H_2 \uplus H_3 \uplus S, B, T\rangle_n \rightarrowtail_{apply} \\ &\qquad \langle C \uplus no\_id(A), \{c\#i\} \uplus H_1 \uplus H_2 \uplus S, \theta \wedge B, T'\rangle_n\end{aligned}$$

Now $\langle C \uplus no\_id(A), \{c\#i\} \uplus H_1 \uplus H_2 \uplus S, \theta \wedge B, T'\rangle_n = \alpha(\sigma_{i+1})$, so $\alpha(\sigma_i) \rightarrowtail_{apply} \alpha(\sigma_{i+1})$ where the ω_t transition is **Apply**. Hence Case 2 above is satisfied.
CASE **Default**: $\sigma_i \rightarrowtail \sigma_{i+1}$ is of the form

$$\langle [c\#i : j|A], S, B, T\rangle_n \rightarrowtail_{default} \langle [c\#i : j+1|A], S, B, T\rangle_n$$

Then $\alpha(\sigma_i) = \alpha(\sigma_{i+1}) = \langle no\_id(A), S, B, T\rangle_n$. Hence Case 1 above is satisfied.
Therefore, for all ω_r derivations D, $\alpha(D)$ is a ω_t derivation. □

We have shown that every ω_r derivation has a corresponding ω_t derivation given by function α. But in order to show the correspondence of the semantics we also need to show that the terminating ω_r derivations map to terminating ω_t derivations. First we need to define what subset of all ω_r execution states need to be considered.

Definition 13 (Reachability) *An execution state σ (of either semantics) is* reachable *if there exists an initial state $\sigma_0 = \langle G, \emptyset, true, \emptyset\rangle_1$ such that there exists a derivation $\sigma_0 \rightarrowtail^* \sigma$.*

Not all states are reachable.

Example 10 *The following execution state is not reachable with respect to the* `gcd` *program from Example 1.*

$$\langle [], \{gcd(0)\#1\}\rangle_2$$

If a `gcd(0)` *constraint is present in the store, then at some stage in the derivation that constraint must have been active. When it was active the first rule must have fired, hence the constraint must have been deleted. Therefore the above program state is not reachable.*

Reachability is generally undecidable for programming languages and this certainly applies to CHRs.

Before we state the main theorem we need to prove two lemmas related to reachability of ω_r states, i.e. given a state of a specified form, together with some additional assumptions, can we find a future state in the derivation that matches some other specified form.

The first lemma says that if we have some prefix of constraints on the top of the execution stack, then eventually those constraints will be removed assuming termination and non-failure.

Lemma 1 (Intermediate States 1) *Let D be a finite ω_r derivation from an execution state σ of the form $\langle A_p +\!\!+ A_s, S, B, T\rangle_n$ (for non-empty A_s) to some non-$false$ final state. Then there exists an intermediate state σ_k of the form $\langle A_s, S_k, B_k, T_k\rangle_{n_k}$ (the same A_s) in D.*

Proof
Direct proof. We define an abstraction function β which takes a suffix A'_s and a state $\langle A'_p +\!\!+ A'_s, S', B', T'\rangle_{n'}$ where the execution stack ends with the suffix A'_s, and returns the length of the prefix A'_p.

$$\beta(A'_s, \langle A'_p +\!\!+ A'_s, S', B', T'\rangle_{n'}) = len(A'_p)$$

Where function len returns the length of a sequence defined in the standard way. The function $\beta(A'_s, \sigma')$ is undefined if the execution stack of σ' does not end with A'_s.

The function $\beta(A_s, \sigma)$ is well-defined for suffix A_s and initial state σ from above. Let σ_f be the final state in derivation D. Since σ_f is non-$false$, it must be of the form $\langle [], S_f, B_f, T_f\rangle_{n_f}$. As A_s is non-empty by assumption, function $\beta(A_s, \sigma_f)$ is undefined for σ_f. Therefore there must exist two consecutive states σ_i and σ_{i+1} in D such that $\beta(A_s, \sigma_i)$ is defined but $\beta(A_s, \sigma_{i+1})$ is not.

We consider the possible transitions between σ_i and σ_{i+1}.
CASE **Activate**, **Reactivate** and **Default**: If $\beta(A_s, \sigma_i)$ is defined then $\beta(A_s, \sigma_{i+1}) = \beta(A_s, \sigma_i)$ is also defined. Hence we can exclude these cases.
CASE **Drop**: If $\beta(A_s, \sigma_i)$ is defined then $\beta(A_s, \sigma_{i+1}) = \beta(A_s, \sigma_i) - 1$ is defined if $\beta(A_s, \sigma_i) \geq 1$, otherwise $\beta(A_s, \sigma_{i+1})$ is undefined.
CASE **Solve**: If $\beta(A_s, \sigma_i)$ is defined then $\beta(A_s, \sigma_{i+1}) = \beta(A_s, \sigma_i) + len(S_1) - 1$ (where S_1 represents that constraints woken up by the wakeup policy) is defined if $len(S_1) \geq 1$ or $\beta(A_s, \sigma_i) \geq 1$, otherwise $\beta(A_s, \sigma_{i+1})$ is undefined.
CASE **Simplify**: If $\beta(A_s, \sigma_i)$ is defined then $\beta(A_s, \sigma_{i+1}) = \beta(A_s, \sigma_i) + len(C) - 1$ (where C is the body of the rule that fired) is always defined since $len(C) \geq 1$ (the body of a rule must be at least of length 1). Hence we can exclude this case.
CASE **Propagate**: If $\beta(A_s, \sigma_i)$ is defined then $\beta(A_s, \sigma_{i+1}) = \beta(A_s, \sigma_i) + len(C)$ (where C is the body of the rule that fired) is always defined. Hence we can exclude this case.

Hence the only possible transitions between σ_i and σ_{i+1} are

1. **Drop** when $\beta(A_s, \sigma_i) = 0$; and
2. **Solve** when $\beta(A_s, \sigma_i) = 0$ and $len(S_1) = 0$.

In either case, $\beta(A_s, \sigma_i) = 0$ holds iff σ_i is of the form $\langle A_s, S_i, B_i, T_i\rangle_{n_i}$, hence $\sigma_k = \sigma_i$ directly satisfies our hypothesis. □

The second lemma says that if we have a newly activated constraint (with occurrence number 1), and some occurrence k in the program for that constraint, then the active constraint will eventually reach occurrence k assuming termination, non-failure and the active constraint is never deleted.

Lemma 2 (Intermediate States 2) *Let $\sigma = \langle [c\#i : 0|A], \{c\#i\} \uplus S, B, T\rangle_n$ be an ω_r execution state and D a derivation from σ to some non-$false$ final state σ_f such that $c\#i \in S_f$. Then for all programs P and occurrences k of predicate c in P, there exists an intermediate state σ_k of the form $\langle [c\#i : k|A], \{c\#i\} \uplus S_k, B_k, T_k\rangle_{n_k}$ in D.*

Proof
By induction.
Base case: $k = 1$. Then $\sigma_k = \sigma$ is of the appropriate form.
Induction Step: Assume that for all programs P and occurrences k of predicate c in P, there exists an intermediate state σ_k of the form $\sigma_k = \langle [c\#i : k|A], \{c\#i\} \uplus S_k, B_k, T_k\rangle_n$. We show how to find state $\sigma_{k+1} = \langle [c\#i : k{+}1|A], \{c\#i\}\uplus S_{k+1}, B_{k+1}, T_{k+1}\rangle_{n_{k+1}}$.

Note that because derivation D is finite, there can only be a finite number of states of the form σ_k, so w.l.o.g assume σ_k is the last state in D of the required form.

We consider all possible ω_l transitions applicable to σ_k.
CASE **Solve**, **Activate** and **Reactivate**: None of these are applicable to a state of the form σ_k;
CASE **Drop**: Not applicable since this means occurrence k does not exist, which violates the induction hypothesis;
CASE **Simplify**: This will delete the active constraint which violates our assumption that $c\#i$ is never deleted throughout derivation D.
CASE **Propagate**: Then

$$\langle [c\#i : k|A], \{c\#i\}\uplus S_k, B_k, T_k\rangle_n \rightarrowtail_{propagate} \langle C +\!\!+ [c\#i : k|A], \{c\#i\}\uplus S'_k, B'_k, T'_k\rangle_n$$

We can now apply Lemma 1 to find a future state σ'_k in derivation D of the form

$$\langle [c\#i : k|A], \{c\#i\} \uplus S''_k, B''_k, T''_k\rangle_{n''}$$

But σ'_k is in the same form as σ_k, which contradicts our assumption that σ_k was the last state in the derivation of that form.

Thus the only transitions applicable to σ_k is **Default**.

$$\langle [c\#i : k|A], \{c\#i\} \uplus S_k, B_k, T_k\rangle_n \rightarrowtail_{default} \langle [c\#i : k+1|A], \{c\#i\} \uplus S_k, B_k, T_k\rangle_n$$

The new state is of the required form for $k + 1$.

Therefore for all programs P and occurrences k of predicate c in P, there exists an intermediate state σ_k of the form $\langle [c\#i : k|A], \{c\#i\} \uplus S_k, B_k, T_k\rangle_{n_k}$ in D, provided D starts with state a state of the form $\langle [c\#i : 1|A], \{c\#i\} \uplus S, B, T\rangle_n$, and does not delete $c\#i$. $\square$

We can now show that reachable final ω_r states map to reachable[8] final states under ω_t.

Theorem 2 (Final States) *Let σ be a reachable final state under ω_r, then $\alpha(\sigma)$ is a reachable final state under ω_t.*

Proof
By contradiction. Assume that $\alpha(\sigma)$ is not a final state, i.e. $\alpha(\sigma)$ can fire at least one ω_t transition.

Because σ is a *reachable* final state, these exists an initial state σ_0 such that

[8] Reachability is obviously preserved thanks to Theorem 1

$\sigma_0 \rightarrowtail_D^* \sigma$ for some ω_r derivation D. For convenience, we rename $\sigma = \sigma_f$ and name every state in the derivation $\sigma_i = \langle A_i, S_i, B_i, T_i \rangle_{n_i}$ for some i such that $D = \sigma_0 \rightarrowtail \sigma_1 \rightarrowtail ... \rightarrowtail \sigma_f$. By Theorem 1, $\alpha(D)$ is a ω_t derivation from $\alpha(\sigma_0)$ to $\alpha(\sigma_f)$.

Execution state σ_n is a final state, therefore $\sigma_n = \langle [], S_f, B_f, T_f \rangle_{m_f}$ (i.e. the execution stack A_f is empty). The other possibility is $\sigma_n = false$, but then $\alpha(\sigma_n) = false$ which is also a final state. Hence $\alpha(\sigma_n) = \langle \emptyset, S_f, B_f, T_f \rangle_{m_f}$, and only the **Apply** transition is applicable to such a ω_t state (the goal field is empty).

Let $H_1 \uplus H_2 \subseteq S_f$, and let $(r \ @ \ H_1' \setminus H_2' \iff g \mid C)$ be the instance of the rule that matches with $\theta \equiv chr(H_1) = H_1' \wedge chr(H_2) = H_2'$. We also know that $\mathcal{D} \models B_f \rightarrow \exists_r(\theta \wedge g)$, and $id(H_1) +\!\!+ \, id(H_2) +\!\!+ \, [r] \notin T_f$ otherwise **Apply** is not applicable.

Consider the derivation D. The built-in store is monotonically increasing throughout the derivation, i.e. for all built-in stores B_i and B_j from states $\sigma_i, \sigma_j \in D$, if $i < j$ then $B_i \subseteq B_j$ (by treating B_i and B_j as multisets). This is easily verified by observing none of the ω_r derivations delete constraints from the built-in store. If for some σ_i in D we have that $\mathcal{D} \models B_i \rightarrow \exists_r(\theta \wedge g)$, then for all $j > i$ we have that $\mathcal{D} \models B_j \rightarrow \exists_r(\theta \wedge g)$.

For final state σ_f, the guard holds, i.e. $\mathcal{D} \models B_f \rightarrow \exists_r(\theta \wedge g)$. Therefore there must exist a *first* state $\sigma_b \in D$ such that the guard holds, i.e. $\mathcal{D} \models B_b \rightarrow \exists_r(\theta \wedge g)$, but not for σ_j where $j < b$, i.e. $\mathcal{D} \not\models B_j \rightarrow \exists_r(\theta \wedge g)$. Note the index '$b$' in σ_b stands for "$b$uilt-in" – the first state where the built-in store entails the guard g.

Let σ_s be the *first* execution state in D where $H_1 \uplus H_2 \subseteq S_s$ where S_s is the CHR store of σ_s. Such a state must exist because $H_1 \uplus H_2 \subseteq S_f$, where S_f is the CHR store of final state σ_f. Here the 's' stands for "CHR $s$tore" – the first state where all constraints $H_1 \uplus H_2$ are in the CHR store.

We know that in derivation D, the states σ_b and σ_s must be present. There are two cases we need to consider, namely $b \leq s$ and $b > s$.

CASE $b \leq s$: Let $c\#i \in H_1 \uplus H_2$ be the CHR constraint such that $c\#i \in S_s$ (the store for σ_s) but $c\#i \notin S_{s-1}$ (the store for σ_{s-1}), i.e. $c\#i$ is the last constraint in $H_1 \uplus H_2$ to be added to the store. The ω_r transition between σ_{s-1} and σ_s must be **Activate**, since this is the only transition that will add a CHR constraint into the store. **Activate** also activates the topmost constraint, so for some A_s' and S_s' we have $\sigma_s = \langle [c\#i : 1 | A_s'], \{c\#i\} \uplus S_s', B_s, T_s \rangle_{n_s}$.

Let k be the occurrence of predicate c in rule r (from above) that matched with c when we applied the **Apply** transition to $\alpha(S_f)$. By Lemma 2 there exists at least one future state σ_t of the form $\langle [c\#i : k | A_t'], \{c\#i\} \uplus S_t', B_t, T_t \rangle_{n_t}$. Since D is finite, there must be a *last* state in the form σ_t, so w.l.o.g. assume that σ_t is the last state in D of the above form. The only possible applicable transition that doesn't violate one of our assumptions is **Propagate**, because:

1. **Solve**, **Activate**, **Reactivate** and **Drop** are directly not applicable to a state in the form of σ_t;
2. **Default** is not applicable, since we know this state could potentially fire the **Propagate** transition on rule r because $H_1 \uplus H_2 \subseteq S_t$ matches against the head of rule

r with θ (where θ is exactly the same as the one from above). We know that $\mathcal{D} \models B_t \rightarrow \exists_r(\theta \wedge g)$ because of the assumption $b \leq s$ and $id(H_1) +\!\!+ \; id(H_2) +\!\!+ \; [r] \not\in T_t$ because no such entry appears in the final state;

3. **Simplify** is not applicable since it will delete the active $c\#i$ constraint, violating our assumption that $c\#i$ appears in the final store of σ_f.

So the transition applied to σ_t in D must be **Propagate** (on a different matching).

$$\langle [c\#i : k|A'_t], \{c\#i\} \uplus S'_t, B_t, T_t \rangle_{n_t} \rightarrowtail_{propagate} \\ \langle C +\!\!+ \; [c\#i : k|A'_t], \{c\#i\} \uplus S'_t, B'_t, T'_t \rangle_{n_t}$$

We can now apply Lemma 1 to find a future state σ_u in derivation D of the form $\langle [c\#i : k|A'_t], \{c\#i\} \uplus S'_u, B_u, T_u \rangle_{n_u}$. But σ_u is in the same form as σ_t which contradicts our assumption that σ_t was the last such state.

CASE $b > s$:

Consider the state σ_{b-1}, i.e. the state just before the built-in store satisfies the guard g.

The transitions that modify the built-in constraint store are **Solve**, **Simplify** and **Propagate**. We can exclude both **Simplify** and **Propagate**, since these only introduce equations on fresh variables (i.e. the θ) which does not change the meaning of the store. Hence **Solve** must be the transition between σ_{b-1} and σ_b. By the *lower bound* condition of a wakeup policy used by **Solve**, there must be a constraint $c\#i \in H_1 \uplus H_2$ such that $c\#i$ is an element of the constraints woken up by the wakeup policy. Therefore σ_b must be of the form $\langle A_p +\!\!+ \; [c\#i|A_s], \{c\#i\} \uplus S'_b, B_b, T_b \rangle_{n_b}$. In other words, after applying **Solve** on σ_{b-1}, the constraint $c\#i$ must appear somewhere on the execution stack.

We can now apply Lemma 1 to derive a future state σ'_b of the form $\langle [c\#i|A_s], \{c\#i\} \uplus S''_b, B'_b, T'_b \rangle_{n'_b}$. The only transition applicable to such a state is **Reactivate**, hence

$$\langle [c\#i|A_s], \{c\#i\} \uplus S''_b, B'_b, T'_b \rangle_{n'_b} \rightarrowtail_{reactivate} \langle [c\#i : 1|A_s], \{c\#i\} \uplus S''_b, B'_b, T'_b \rangle_{n'_b}$$

Now this new state is in the same form as σ_s from the $b \leq s$ case (see above), hence we can apply the same argument as before to derive the same contradiction. □

Theorem 1 and Theorem 2 show that the refined operational semantics correctly implement the theoretical operational semantics.

4.1 Termination

Termination of CHR programs is obviously a desirable property. Thanks to Theorems 1 and 2, termination of ω_t programs ensures termination of ω_r.

Firstly we need to show that all ω_r derivations consisting only of **Reactivate**, **Drop** and **Default** transitions are finite. Notice that these are the transitions that disappear after function α has been applied to a ω_r derivation (see the proof of Theorem 1).

Lemma 3 *Let σ be an ω_r execution state, then there is no infinite ω_r derivation $\sigma \rightarrowtail^{\infty}$ consisting of only* ***Reactivate****,* ***Drop*** *and* ***Default*** *transitions.*

Proof
By constructing a well founded order over such derivations. Firstly define a ranking (abstraction) function *rank* that maps ω_r execution states and a CHR program P to a triple of non-negative integers.

$$rank(\langle A, S, B, T\rangle_i, P) = (len(A), len(nums(A)), total(P) - occ(A))$$

Where function *len* maps a sequence to the length of that sequence, defined in the standard way. Function *nums* maps a sequence of constraints to a sequence of numbered constraints by filtering out all non-numbered and active constraints.

$$\begin{array}{ll} nums([]) & = [] \\ nums([c|A]) & = nums(A) \\ nums([c\#i|A]) & = [c\#i] +\!\!+\ nums(A) \\ nums([c\#i : j|A]) & = nums(A) \end{array}$$

Function *occ* maps a sequence of constraints A to the occurrence number of the top-most active constraint if it exists; or 0 otherwise.

$$\begin{array}{ll} occ([]) & = 0 \\ occ([c|A]) & = 0 \\ occ([c\#i|A]) & = 0 \\ occ([c\#i : j|A]) & = j \end{array}$$

Finally function *total* maps a program P to the total number of constraints in the heads of every rule plus one.

$$\begin{array}{ll} total([]) & = 1 \\ total([(r \ @ \ H_1 \setminus H_2 \iff g \mid C)|P]) & = len(H_1) + len(H_2) + total(P) \end{array}$$

Let $\prec$ be the standard lexicographical tuple ordering. For all reachable execution states σ and programs P, $rank(\sigma, P) \succeq (0, 0, 0)$. This directly follows from the fact that $len(A) \geq 0$ for all sequences A, and $total(P) \geq occ(A)$ for all sequences of constraints A and programs P. Thus ordering over the ranks of execution states is *well-founded*, i.e. no infinite decreasing chains of execution state rankings $rank(\sigma_0, P) \succ rank(\sigma_1, P) \succ ....$

Next we show that for all execution states σ and σ' such that $\sigma \rightarrowtail \sigma'$ by transition **Reactivate**, **Drop** or **Default**, then $rank(\sigma', P) \prec rank(\sigma, P)$.
CASE **Drop**: $\sigma \rightarrowtail \sigma'$ is of the form

$$\langle [c\#i : j|A], S, B, T\rangle_n \rightarrowtail_{drop} \langle A, S, B, T\rangle_n$$

If $rank(\sigma, P) = (x_1, x_2, x_3)$, then $rank(\sigma', P) = (x_1 - 1, x_2, x'_3)$ for some x'_3. Hence $rank(\sigma', P) \prec rank(\sigma, P)$.
CASE **Reactivate**: $\sigma \rightarrowtail \sigma'$ is of the form

$$\langle [c\#i|A], S, B, T\rangle_n \rightarrowtail_{reactivate} \langle [c\#i : 1|A], S, B, T\rangle_n$$

If $rank(\sigma, P) = (x_1, x_2, x_3)$, then $rank(\sigma', P) = (x_1, x_2 - 1, x'_3)$ for some x'_3. Hence $rank(\sigma', P) \prec rank(\sigma, P)$.
CASE **Default**: $\sigma \rightarrowtail \sigma'$ is of the form

$$\langle [c\#i : j|A], S, B, T\rangle_n \rightarrowtail_{default} \langle [c\#i : j + 1|A], S, B, T\rangle_n$$

If $rank(\sigma, P) = (x_1, x_2, x_3)$, then $rank(\sigma', P) = (x_1, x_2, x_3{-}1)$. Hence $rank(\sigma', P) \prec rank(\sigma, P)$.

Thus we have established a termination order, thus proving derivations consisting of only **Reactivate**, **Drop** and **Default** must be finite. $\square$

We can now state the main termination result.

Lemma 4 *Let σ_0 be an ω_r execution state. If every derivation for $\alpha(\sigma_0)$ terminates under ω_t, then every derivation for σ_0 also terminates under ω_r.*

Proof
By contradiction. Assume $\alpha(\sigma_0)$ terminates under ω_t, but not for σ_0 with respect to ω_r. Then there exists an infinite ω_r derivation D starting from σ_0. By Theorem 1 there must be a corresponding derivation $\alpha(D)$ from initial state $\alpha(\sigma_0)$ with respect to ω_t. By assumption, $\alpha(D)$ must be finite.

We partition derivation D into infinitely many subderivations $D_0 ++ D_1 ++ D_2 ++ ...$ as follows. Each D_i is a finite sub-derivation of D starting from the last state in D_{i-1} for $i > 0$ (or σ_0 otherwise) to a state σ_i such that the transition between the state proceeding σ_i and σ_i in D is either **Solve**, **Activate**, **Simplify** or **Propagate**. All other transitions in D_i must be **Reactivate**, **Drop** and **Default**. In other words, $D_i = D'_i \rightarrowtail_i \sigma_i$ where D'_i is a (possibly trivial) subderivation of D consisting only of **Reactivate**, **Drop** and **Default** transitions, and transition $\rightarrowtail_i$ to state σ_i is either **Solve**, **Activate**, **Simplify** or **Propagate**.

Note that it is always possible to partition D in this way. Otherwise suppose that $D = D_0 ++ D_1 ++ D_2 ++ ... ++ D_n ++ D'$ where it is not possible to further partition D', then D' must not contain a **Solve**, **Activate**, **Simplify** or **Propagate** transition. Then Lemma 3 implies D' is finite, thus D is finite, which directly contradicts our initial assumption that D is infinite. So $D = D_0 ++ D_1 ++ D_2 ++ ...$ for infinitely many D_i.

Now $\alpha(D) = \alpha(D_0 ++ D_1 ++ D_2 ++ ...) = \alpha(D_0) ++ \alpha(D_1) ++ \alpha(D_2) ++ ....$ Each D_i contains one **Solve**, **Activate**, **Simplify** or **Propagate** transition, hence each $\alpha(D_i)$ has non-zero length (see the proof of Theorem 1). As the length of $\alpha(D)$ is the sum of the (non-zero) lengths of every $\alpha(D_i)$, and there are infinitely many $\alpha(D_i)$, then $\alpha(D)$ has infinite length which is a contradiction. $\square$

The converse is clearly not true, as shown in Example 1.

In practice, proving termination for CHR programs under the theoretical operational semantics is quite difficult (see (Frühwirth 1999) for examples and discussion). It is somewhat simpler for the refined operational semantics but, just as with other programming languages, this is simply left to the programmer.

4.2 Confluence

Both operational semantics for CHRs are nondeterministic, therefore the property of *confluence* (which guarantees the same result no matter the order transitions are applied) is essential from a programmer's point of view. Without it the programmer cannot anticipate the answer that will arise from a goal.

There are two main sources of nondeterminism from the refined semantics. The first is from the **Solve** transition, where the order (and number of repeats) of the woken up constraints are (re)added to the execution stack is unspecified. The second is from the **Simplify** and **Propagate** transitions, where there may be more than one choice for choosing matching partner constraints.

Example 11 *Consider a CHR implementation of a simple database:*

```
l1 @ entry(Key,Val)₁ \ lookup(Key,ValOut)₁ <=> ValOut = Val.
l2 @ lookup(_,_)₂ <=> fail.
```

where the constraint `lookup` *represents the basic database operations of key lookup, and* `entry` *represents a piece of data currently in the database (an entry in the database). Rule* `l1` *looks for the matching entry to a lookup query and returns in* `ValOut` *the stored value. Rule* `l2` *causes a lookup to fail if there is no matching entry.*

Consider the following (simplified) execution state with two database entries for the same key, and an active `lookup` *constraint on the stack.*

$$\langle[\texttt{lookup}(key, V)\#3 : 1], \{\texttt{lookup}(key, V)\#3, \texttt{entry}(key, cat)\#2, \texttt{entry}(key, dog)\#1\}, true\rangle_4$$

Now the active `lookup` *constraint is at occurrence* 1 *from rule* `l1` *above, so* ***Simplify*** *is applicable matching against either* $\texttt{entry}(key, cat)$ *or* $\texttt{entry}(key, dog)$ *from the store. Depending on what matching is chosen (both are equally valid), the resulting state after* ***Simplify*** *is*

$$\langle[], \{\texttt{entry}(key, cat)\#2, \texttt{entry}(key, dog)\#1\}, V = cat\rangle_4$$

or

$$\langle[], \{\texttt{entry}(key, cat)\#2, \texttt{entry}(key, dog)\#1\}, V = dog\rangle_4$$

Since these are both final states are not the same result, it follows that the `database` *program is non-confluent.*

In order to properly formalise the property of confluence first we need to define what it means to get the "same result". Unfortunately, a straightforward syntactic comparison is too strong in general, since we do not care about constraint numbering and similar things. We do however care about propagation histories, because "equivalent" states should be similarly applicable to the same set of rules. We define a mapping which extracts the part of a propagation history that we care about as follows.

Definition 14 (Live History) *Function alive is a bijective mapping from a CHR* *store S and a propagation history to a propagation history defined as follows.*

$$\begin{array}{lll} alive(S,\emptyset) & = \emptyset & \\ alive(S,\{t\} \uplus T) & = alive(S,t) \uplus alive(S,T) & \\ alive(S, t +\!\!+ [\_]) & = \emptyset & \textit{if } \exists i \in t \textit{ such that } \forall c(c\#i \notin S) \\ alive(\_, t) & = \{t\} & \textit{otherwise} \end{array}$$

In other words, $alive(S,T)$ is propagation history T where all entries with numbers for deleted (i.e. not alive) constraints have been removed. Note that $alive(S,T)$ can only have entries on propagation rules (otherwise one of the numbers in the entry must be dead).

We can now formally define variance between two states.

Definition 15 (Variants) *Two states*

$$\sigma_1 = \langle A_1, S_1, B_1, T_1\rangle_{i_1}^{\mathcal{V}} \qquad \textit{and} \qquad \sigma_2 = \langle A_2, S_2, B_2, T_2\rangle_{i_2}^{\mathcal{V}}$$

(from either semantics) are variants *if there exists a renaming ρ on variables not in $\mathcal{V}$ and a mapping ϱ on constraint numbers such that*

1. $\rho \circ \varrho(A_1) = A_2$ *(sequence equality for ω_r, multiset equality for ω_t);*
2. $\rho \circ \varrho(S_1) = S_2$;
3. $\mathcal{D} \models (\bar{\exists}_{\mathcal{V}}\rho(B_1) \leftrightarrow \bar{\exists}_{\mathcal{V}}B_2)$; *and*
4. $\varrho \circ alive(S_1,T_1) = alive(S_2,T_2)$.

Otherwise the two states are variants if $\mathcal{D} \models \neg\bar{\exists}_{\emptyset}B_1$ and $\mathcal{D} \models \neg\bar{\exists}_{\emptyset}B_2$ (i.e. both states are $false$).

In other words, we consider two (non-$false$) states σ_1 and σ_2 to be variants (i.e. the "same result") if the two goals (or execution stacks) and CHR stores are the same, the built-in solver can prove that the built-in stores are logically equivalent, and the "live" parts of the propagation histories are the same, all modulo variables not appearing in $\mathcal{V}$ and constraint numbering.

We can now define *joinability*, which is the property that two execution states reduce to the same answer.

Definition 16 (Joinable) *Two states σ_1 and σ_2 are* joinable *if there exists states σ_1' and σ_2' such that $\sigma_1 \rightarrowtail^* \sigma_1'$ and $\sigma_2 \rightarrowtail^* \sigma_2'$ and σ_1' and σ_2' are variants.*

Now we can formally define confluence as follows.

Definition 17 (Confluence) *A CHR program P is* confluent *with respect to operational semantics ω if the following holds for all states σ_0, σ_1 and σ_2 where σ_0 is a reachable state: If $\sigma_0 \rightarrowtail_{\omega}^* \sigma_1$ and $\sigma_0 \rightarrowtail_{\omega}^* \sigma_2$ then σ_1 and σ_2 are joinable with respect to σ_0.*

This definition is slightly stronger than the classical definition in (Abdennadher 1997) since we require σ_0 to be a *reachable* state. This is important, since the programmer generally only cares about reachable states. Our definition is stronger, so confluence under the classical definition implies confluence under our new definition.

Example 12 *The* `gcd` *program from Example 1 is confluent under both operational semantics (although it may not terminate under the theoretical semantics). This is because any final state derived from the initial goal* $\texttt{gcd}(i_1), ..., \texttt{gcd}(i_n)$ *must contain only the constraint* $\texttt{gcd}(gcd(i_1, ..., i_n))$ *in the store. The built-in stores must also be equivalent once all temporary variables (created by matching constraints against rules) are renamed. Hence the* `gcd` *program is confluent.*

Confluence of the theoretical operational semantics of CHR programs has been extensively studied (Frühwirth 1998; Abdennadher 1997; Abdennadher et al. 1999). Abdennadher (1997) provides a decidable confluence test for the theoretical semantics of terminating CHR programs. Essentially, it relies on computing critical pairs where two rules can possibly be used, and showing that each of the two resulting states lead to variant states.

Just as with termination, confluence under ω_t implies confluence under ω_r provided the program also terminates under ω_r.

Corollary 1 *If CHR program P is terminating under ω_r, and confluent under ω_t, then it is also confluent under ω_r.*

Proof
By contradiction. Assume that P is confluent under ω_t, but not confluent with respect to ω_r. Then there exists a reachable ω_r state σ_0 such that $\sigma_0 \rightarrowtail^* \sigma_1'$ and $\sigma_0 \rightarrowtail \sigma_2'$ where σ_1' and σ_2' are not joinable. Let σ_1 and σ_2 be final states derived from σ_1' and σ_2' respectively, i.e.

$$\begin{array}{l} \sigma_1' \rightarrowtail^* \sigma_1 = \langle A_1, S_1, B_1, T_1 \rangle_{i_1}^{\mathcal{V}} \\ \sigma_2' \rightarrowtail^* \sigma_2 = \langle A_2, S_2, B_2, T_2 \rangle_{i_2}^{\mathcal{V}} \end{array}$$

Where $A_1 = A_2 = []$ or both σ_1 and σ_2 are $false$. Note that it always possible to find σ_1 and σ_2 because of the assumption of termination under ω_r. As σ_1' and σ_2' are not joinable both σ_1 and σ_2 cannot be variants. Note also that by construction $\sigma_0 \rightarrowtail^* \sigma_1$ and $\sigma_0 \rightarrowtail^* \sigma_2$.

By Theorem 1, there exists two derivations under ω_t:

$$\begin{array}{l} \alpha(\sigma_0) \rightarrowtail^* \alpha(\sigma_1) = \langle no\_id(A_1), S_1, B_1, T_1 \rangle_{i_1}^{\mathcal{V}} \\ \alpha(\sigma_0) \rightarrowtail^* \alpha(\sigma_2) = \langle no\_id(A_2), S_2, B_2, T_2 \rangle_{i_2}^{\mathcal{V}} \end{array}$$

By assumption P is confluent under the theoretical operational semantics, therefore both $\alpha(\sigma_1)$ and $\alpha(\sigma_1)$ must be joinable. By Theorem 2 both $\alpha(\sigma_1)$ and $\alpha(\sigma_2)$ are final states, therefore to be joinable they must be variants.

There are two cases to consider.
CASE 1: $\mathcal{D} \models \neg\bar{\exists}_\emptyset B_1$ and $\mathcal{D} \models \neg\bar{\exists}_\emptyset B_2$:
(I.e. both states are $false$). Then σ_1 and σ_2 must be variants since the built-in store is unaffected by abstraction function α.
CASE 2: There exists a renaming ρ on variables not in σ_0 and a mapping ϱ on constraint numbers such that $\rho \circ \varrho(S_1) = S_2$, $\mathcal{D} \models (\bar{\exists}_\mathcal{V}\rho(B_1) \leftrightarrow \bar{\exists}_\mathcal{V}B_2)$ and $\varrho \circ alive(S_1, T_1) = alive(S_2, T_2)$. Note that because σ_1 and σ_2 are final states, their

execution stacks are empty, i.e. $A_1 = A_2 = []$. Therefore σ_1 and σ_2 are variants by definition.

Both cases directly contradict our assumption that σ_1 and σ_2 are not variants. Therefore if program P which is terminating under ω_r is confluent under ω_t, then it is also confluent under ω_r. $\square$

Example 13 *Both the* `gcd` *and* `leq` *programs (from Example 1 and Example 7) are terminating under ω_r, and are confluent under ω_t. Therefore, by Corollary 1, confluence under ω_r immediately follows.*

The converse of Corollary 1 is not true, as shown by the following simple example.

Example 14 *The following program is confluent under ω_r (since an active* `p` *constraint always fires rule* `r1`*).*

```
r1 @ p <=> true.
r2 @ p <=> false.
```

However the program is not confluent under ω_t (since `p` *can fire either* `r1` *or* `r2` *resulting in non-joinable states).*

This shows that the set of confluent programs under ω_r is larger than the same set under ω_t.

5 Practical Confluence Test

It is common for programmers to write CHR programs that are confluent under the refined semantics, but are not confluent under the theoretical semantics. Corollary 1 is not useful for such programs. In this section we discuss a more practical confluence test purely for the refined semantics.

There are two sources of nondeterminism under the refined operational semantics. The first arises from the **Solve** transition, where the order in which the woken up constraints are (re)added to the execution stack is left open.

Example 15 *Consider the following state for the* `leq` *program.*

$$\langle [C = B], \{\mathtt{leq}(C, A)\#2, \mathtt{leq}(A, B)\#1\}, true\rangle_5$$

Assume that the wakeup policy includes all non-fixed CHR constraints. ***Solve*** *requires constraints $\mathtt{leq}(C, A)$ and $\mathtt{leq}(A, B)$ to be (re)added to execution stack. The order is arbitrary, hence*

$$\langle [\mathtt{leq}(C, A)\#2, \mathtt{leq}(A, C)\#1], \{\mathtt{leq}(C, A)\#2, \mathtt{leq}(A, B)\#1\}, C = B\rangle_5$$

or

$$\langle [\mathtt{leq}(A, C)\#1, \mathtt{leq}(C, A)\#2], \{\mathtt{leq}(C, A)\#2, \mathtt{leq}(A, B)\#1\}, C = B\rangle_5$$

are equally valid states after applying ***Solve****.*

In this example the choice of the states after **Solve** is inconsequential, because the `leq` program is confluent under the refined semantics.

The other source of nondeterminism arises the **Simplify** and **Propagate** transitions, which do not specify which partner constraints (i.e. H_1, H_2 and H_3 from Definition 10) should be chosen for the transition (if more than one possibility exists). Example 11 shows how this choice can result in two non-variant final states.

Both sources of nondeterminism could be removed by further "refining" the operational semantics. For example, we could impose an order on matchings for **Simplify** and **Propagate**, or an order on the constraints woken up after **Solve**. The advantage is that all programs are trivially confluent under a deterministic operational semantics.

There are two main reasons against this idea. The first is that different CHR implementations use different data structures to (efficiently) represent the store, and this usually affects the order partner constraints are matched against the head of a rule. By imposing an artificial order on partner constraints may have an adverse effect on efficiency, since we have restricted or complicated the data structures that can be used. The second reason is that it not clear how further refining the semantics benefits the programmer. Many CHR programs have already been implemented using the refined operational semantics without any additional assumptions about orderings, etc. Therefore if a program is not confluent under the refined semantics then this generally indicates a bug, and ideally the compiler should detect this if possible.

We provided a theoretical result in Section 4.1 which ensures that terminating (under ω_r) and confluent (under ω_t) programs are confluent under the refined semantics. Sometimes this is useful, e.g. with the `leq` program, but in general programs under the refined semantics are not confluent under the theoretical semantics. This directly follows from the fact that the refined semantics is more deterministic. In this section we look at testing for confluence under the refined semantics alone. We propose several static analyses designed to detect non-confluent programs. A confluence checker has been implemented as part of the new Mercury CHR compiler,[9] and we test it on several "large" CHR programs.

5.1 Nondeterminism in the Solve transition

The first source of nondeterminism under the refined semantics occurs when deciding the order on the the set of woken up constraints during a **Solve** transition. To avoid this nondeterminism we will require this set to be empty.[10] This is a very common case, as it occurs when the all CHR constraints are fixed/ground at runtime. We will generalise this slightly by imposing conditions on the wakeup policy used by the implementation.

[9] The Mercury CHR compiler is the successor of the old HAL CHR compiler (Holzbaur et al. 2001) which is no longer maintained.
[10] Another possibility is to require this set to be singleton.

To formalise this we define the *trivial wakeup policy* that does not wakeup any CHR constraint on a **Solve** transition.

Definition 18 (Trivial Wakeup Policy) *Given a CHR store S, built-in constraint c and built-in store B, we define the* trivial wakeup policy *as*

$$trivial(S, c, B) = \emptyset$$

For $trivial(S, c, B)$ to satisfy the definition of a wakeup policy (see Definition 9), the constraints in S must always be fixed.[11] The Mercury CHR compiler determines this information from `mode` declarations, i.e. a constraint will always be fixed if each argument has the declared mode of '`in`'.

For programs that really do interact with a built-in constraint solver (e.g. the `leq` solver from Example 7), we currently have no better test other than relying on the confluence test of the theoretical operational semantics. In this case it is very hard to see how the programmer can control execution sufficiently.

5.2 Nondeterminism in the Simplify and Propagate Transitions

The second source of nondeterminism occurs when there is more than one set of partner constraints in the CHR store that can be matched against when applying the **Simplify** or **Propagate** transitions.

We formalise this as follows. A *matching* is a sequence of numbered constraints from the CHR store that match with the head of a rule when applying **Simplify** or **Propagate**.

Definition 19 (Matching) *A* matching *M of occurrence j with active CHR constraint c in state $\langle [c\#i : j|A], S, B, T\rangle_n$ is a named tuple of numbered constraints from S that match against the head of rule r of occurrence j. These are*

$$\begin{array}{ll} M = prop(H_1, c\#i, H_2, H_3) & \textit{for } \textbf{Propagate} \\ M = simp(H_1, H_2, c\#i, H_3) & \textit{for } \textbf{Simplify} \end{array}$$

where H_1, H_2 and H_3 are the matching constraints as defined by Definition 10 (the refined semantics).

Note that the order of the constraints in a matching M exactly corresponds with the order of the constraints in the rule that matched with it.

Most of the time we will treat matchings as sequences or multisets. For example, the matching $prop(H_1, c\#i, H_2, H_3)$ can be treated as the sequence $(H_1 +\!\!+ [c\#i] +\!\!+ H_2 +\!\!+ H_3)$. Similarly, $simp(H_1, H_2, c\#i, H_3)$ is treated as $(H_1 +\!\!+ H_2 +\!\!+ [c\#i] +\!\!+ H_3)$.

[11] There may be other restrictive circumstances where the usage of $trivial(S, c, B)$ as a wakeup policy is correct.

The definition of the refined operational semantics does not specify which matching to choose if more than one is available. Non-confluence arises when given a state σ, there are more than one possible matchings M_1 and M_2 (w.r.t. some rule r) such that firing r on M_1 results in a different answer than firing r on M_2.

To help simplify things further, we define the following helper functions which map matchings to useful information about matchings. These will be useful later. We define function $delete(M)$ which returns the multiset of constraints in M which are deleted by occurrence, i.e.

$$\begin{array}{l} delete(prop(H_1, c\#i, H_2, H_3)) = H_3 \\ delete(simp(H_1, H_2, c\#i, H_3)) = H_2 \uplus \{c\#i\} \uplus H_3 \end{array}$$

We also define $entry(r, M)$ which returns the propagation history entry associated with a rule r and a matching M, i.e.

$$\begin{array}{l} entry(r, prop(H_1, c\#i, H_2, H_3)) = \\ \qquad ids(H_1) +\!\!+ [i] +\!\!+ ids(H_2) +\!\!+ ids(H_3) +\!\!+ [r] \\ entry(r, simp(H_1, H_2, c\#i, H_3)) = \\ \qquad ids(H_1) +\!\!+ ids(H_2) +\!\!+ [i] +\!\!+ ids(H_3) +\!\!+ [r] \end{array}$$

We also similarly define $\theta(r, M)$ to be the set of equations derived from unifying M with the head of r, and

$$goal(r) = C \qquad\qquad \text{where } r = (H_1 \backslash H_2 \Longleftrightarrow g \mid C)$$

to be the body of the renamed rule used by the transition. Using these functions, we can write the result of applying **Simplify** or **Propagate** to a state $\langle [c\#i : j|A], S, B, T\rangle_n$ and matching $M \subseteq S$ as as $\langle goal(r) +\!\!+ A, S - delete(M), \theta(r, M) \wedge B, \{entry(r, M)\} \cup T\rangle_n$ where r is the renamed copy of the rule used by the transition.

Non-confluence can arise when multiple matchings exist for a rule r, and r is not allowed to eventually try them all. This may happen when firing r with one matching results in the deletion of a constraint in another matching.

Definition 20 (Matching Completeness) *An occurrence j in (renamed) rule r is* matching complete *if for all reachable states $\langle [c\#i : j|A], S, B, T\rangle_n$ with $M_1, ..., M_m$ possible matchings, then for all $M_i \in \{M_1, ..., M_m\}$ if*

$$\langle goal(r), S - delete(M_i), \theta(r, M) \wedge B, \{entry(r, M_i)\} \cup T\rangle_n \rightarrowtail^* \langle A', S', B', T'\rangle_{n'}$$

then for all $M_j \in \{M_1, ..., M_m\} - \{M_i\}$ we have that $M_j \subseteq S'$.

In other words, firing rule r for any matching M_i and executing $goal(r, M_i)$ does not result in the deletion of a constraint occurring in a different matching $M_k, k \neq i$. The intention is that rules will always try all possible matchings unless failure occurs.

Note that r itself may directly delete the active constraint (via the **Simplify** transition). If so, r will only be matching complete if there is only one possible matching, i.e., $m = 1$.

Example 16 *Consider the `database` confluence problem from Example 11. This can be expressed as a matching completeness problem since there exists a state, namely*

$$\langle [\texttt{lookup}(key, V)\#3 : 1], \{\texttt{lookup}(key, V)\#3, \\ \texttt{entry}(key, cat)\#2, \texttt{entry}(key, dog)\#1\}, true\rangle_4$$

with two matchings

$$M_1 = [\texttt{entry}(key, cat)\#2, \texttt{lookup}(key, V)\#3]$$
$$M_2 = [\texttt{entry}(key, dog)\#1, \texttt{lookup}(key, V)\#3]$$

such that firing the rule on M_1 deletes constraint $\texttt{lookup}(key, V)\#3$ *(the active constraint) which also appears in M_2. Therefore the occurrence for `lookup` cannot be matching complete.*

This occurrence will be matching complete if all states where there are multiple matchings for a given lookup are unreachable. This can be achieved by adding a rule that enforces a functional dependency *(Duck and Schrijvers 2005), i.e. a rule that ensures the are no duplicate `entry`/2 constraints on the same key. For example, adding the following rule to the start of the program enforces the appropriate functional dependency.*

```
killdup @ entry(Key,Val1) \ entry(Key,Val2) <=> Val1 = Val2.
```

This rule causes failure if two non-identical entries with the same key appear in the store. Now the occurrence is matching complete, since only one matching will ever be possible.

Matching completeness can also be broken if the body of a rule indirectly deletes constraints from other matchings.

Example 17 *Consider the following CHR program*

```
r1 @ p1, q(X) ==> r(X).
r2 @ p2, r(a) <=> true.
```

The occurrence 1 of `p` in `r1` is not matching complete because of the (reachable) state

$$\langle [\texttt{p}\#3 : 1], \{\texttt{p}\#3, \texttt{q(a)}\#2, \texttt{q(b)}\#1\}\rangle_4$$

with matchings $M_1 = [\texttt{p}\#3, \texttt{q(a)}\#2]$ and $M_2 = [\texttt{p}\#3, \texttt{q(b)}\#1]$. Firing `r1` *against M_1 calls the new constraint* $\texttt{r}(a)$ *which in turn deletes* $\texttt{p}\#3$ *(which appears in both matchings) by firing rule* `r2`. *Therefore occurrence* 1 *for* `p` *is not matching complete.*

A matching complete occurrence is guaranteed to eventually try all possible matchings for a given execution state. However, matching completeness is sometimes too strong if the programmer does not care which matching is chosen. This is common when the rule body does not depend on the matching.

Example 18 *For example, consider the following rule from a simple ray tracer.*

```
shadow @ sphere(C,R,_) \ light_ray(L,P,_,_) <=>
                                        blocks(L,P,C,R) | true.
```

This rule calculates if point P *is in shadow by testing if the ray from light* L *is blocked by a sphere at* C *with radius* R. *Consider an active* light_ray *constraint: there may be more than one* sphere *blocking the ray, however we do not care* which *sphere blocks, just* if *there is a sphere which blocks. This rule is not matching complete, but since the matching chosen does not affect the resulting state, it is matching independent.*

We define *matching independence* as the property that the matching chosen does not matter.

Definition 21 (Matching Independence) *A matching incomplete occurrence for (renamed) rule r that deletes the active constraint only is* matching independent *if for all reachable states $\langle [c\#i : j|A], S, B, T\rangle_n$ with $M_1, \ldots, M_m$ possible matchings, then all of*

$$\langle goal(r), S - delete(M_i), \theta(r, M_i) \wedge B, \{entry(r, M_i)\} \cup T\rangle_n$$

for each $M_i \in \{M_1, ..., M_m\}$ are joinable (see Definition 16).

The rule shadow in Example 18 satisfies the definition since $goal(r) = true$ for all r, i.e. the goal does not depend on the matching chosen.

Suppose that a rule is matching complete, and there are multiple possible matchings. The ordering in which the matchings are tried is still chosen nondeterministically. Hence, there is still potential of non-confluence. For this reason we also require *order independence*, which ensures the choice of order does not affect the result.

Definition 22 (Order Independence) *A matching complete occurrence j in rule r is* order independent *if for all reachable states $\langle [c\#i : j|A], S, B, T\rangle_n$ with $M_1, \ldots, M_m$ possible matchings, the states*

$$\langle goal(r_i), S_j - delete(M_i), \theta(r_i, M_i) \wedge B_j, \{entry(r_i, M_i)\} \cup T_j\rangle_{n_j}$$

and

$$\langle goal(r_j), S_i - delete(M_j), \theta(r_j, M_j) \wedge B_i, \{entry(r_j, M_j)\} \cup T_i\rangle_{n_i}$$

(where r_i and r_j are distinct renamings of r) are joinable for all $M_i, M_j \in \{M_1, \ldots, M_m\}$ where S_i, S_j, B_i, B_j, T_i, T_j, n_i and n_j are given by final states arising from subcomputations

$$\langle goal(r_i), S - delete(M_i), \theta(r_i, M_i) \wedge B, \{entry(r_i, M_i)\} \cup T\rangle_n \rightarrowtail^* \langle A_i, S_i, B_i, T_i\rangle_{n_i} = \sigma_i$$

and

$$\langle goal(r_j), S - delete(M_j), \theta(r_j, M_j) \wedge B, \{entry(r_j, M_j)\} \cup T\rangle_n \rightarrowtail^* \langle A_j, S_j, B_j, T_j\rangle_{n_j} = \sigma_j$$

where σ_i and σ_j are final states.

The following is a typical example of order independence.

Example 19 *Consider the following fragment of code for summing colours from the ray tracer.*

```
add1 @ add_color(C1), color(C2) <=> C3 = C1 + C2, color(C3).
add2 @ add_color(C) <=> color(C).
```

Assume the colours are encoded as ordinary integers (e.g. for a gray scale image). All occurrences of `color` *and* `add_color` *are matching complete. Furthermore, calling* `add_color(`C_1`)`, ..., `add_color(`C_n`)` *results in* `color(`$C_1 + ... + C_n$`)`. *Since addition is associative and commutative, it does not matter in what order the* `add_color` *constraints are called. Consider the occurrence of* `output` *in*

```
render @ output(P) \ light_ray(_,P,C,_) <=> add_color(C).
```

Here, calling `output(`P`)` *calculates the (accumulated) color at point P where any* `light_ray`*s (a ray from a light source) may intersect. If there are multiple light sources, then there may be multiple* `light_ray` *constraints. The order* `add_color` *is called does not matter, hence the occurrence is order independent.*

5.3 Confluence Test

We claim is that if a program P can is shown to satisfy the conditions outlined above, then it is confluent. In this section we present a formal proof of this fact.

Before we present the main result, we prove two useful lemmas.

Lemma 5 (Parallel Derivations I) *For all execution stacks A_1 and A_2 the following holds: $\sigma = \langle G \mathbin{+\!\!+} A_1, S, B, T\rangle_n \rightarrowtail^* \langle G' \mathbin{+\!\!+} A_1, S_k, B_k, T_k\rangle_{n_k} = \sigma_k$ iff $\sigma' = \langle G \mathbin{+\!\!+} A_2, S, B, T\rangle_n \rightarrowtail^* \langle G' \mathbin{+\!\!+} A_2, S_k, B_k, T_k\rangle_{n_k} = \sigma'_k$ or both states σ_k and σ'_k are $false$, provided no states in either derivation is of the form $\langle A_1, S', B', T'\rangle_{n'}$ or $\langle A_2, S', B', T'\rangle_{n'}$ respectively.*

Proof
Note that it suffices to prove one direction of the "iff" only, since the other direction is symmetric (i.e. obtained by substituting A_1 with A_2 and vice-versa). We prove the " $\Longrightarrow$ " direction by induction over derivations of length k.

Base case: Derivations of zero length ($k = 0$). Then $\sigma_0 = \sigma$ and $\sigma'_0 = \sigma$ are zero length derivation of the required form.

Induction step: Assume that for all derivations of length k that if $\sigma = \langle G \mathbin{+\!\!+} A_1, S, B, T\rangle_n \rightarrowtail^* \langle G_k \mathbin{+\!\!+} A_1, S_k, B_k, T_k\rangle_{n_k} = \sigma_k$ then for all A_2 we have that $\sigma' = \langle G \mathbin{+\!\!+} A_2, S, B, T\rangle_n \rightarrowtail^* \langle G_k \mathbin{+\!\!+} A_2, S_k, B_k, T_k\rangle_{n_k} = \sigma'_k$. We show the same holds for derivations of length $k+1$.

We consider all ω_r derivation steps from σ_k to σ_{k+1} and show the same derivation step can be applied to σ'_k to derive σ'_{k+1} of the required form.

By assumption G_k is non-empty, otherwise σ_k is of the form $\langle A_1, S', B', T'\rangle_{n'}$

which is not allowed. Therefore the top-most constraint on the respective execution stacks for σ_k and σ'_k are the same.

The $k+1$ case easily verified by inspection over all of the transition steps for the refined operational semantics (Definition 10). All of these transitions only depend on the top-most constraint of the execution stack, and all transition preserve the tail of the execution stack. Thus, if $\sigma_k \rightarrowtail \sigma_{k+1}$ then $\sigma'_k \rightarrowtail \sigma'_{k+1}$ by the same transition step.

Therefore if $\sigma = \langle G +\!\!+ A_1, S, B, T\rangle_n \rightarrowtail^* \langle G_k +\!\!+ A_1, S_k, B_k, T_k\rangle_{n_k} = \sigma_k$ then for all A_2 we have that $\sigma' = \langle G +\!\!+ A_2, S, B, T\rangle_n \rightarrowtail^* \langle G_k +\!\!+ A_2, S_k, B_k, T_k\rangle_{n_k} = \sigma'_k$ provided the conditions noted in the Lemma above hold. By symmetry the other direction of the "iff" also holds. □

This next Lemma is almost identical to the previous one, except that it handles the case where all of goal G has finished executing.

Lemma 6 (Parallel Derivations II) *For all execution stacks A_1 and A_2 the following holds: $\sigma = \langle G +\!\!+ A_1, S, B, T\rangle_n \rightarrowtail^* \langle A_1, S_k, B_k, T_k\rangle_{n_k} = \sigma_k$ iff $\sigma' = \langle G +\!\!+ A_2, S, B, T\rangle_n \rightarrowtail^* \langle A_2, S_k, B_k, T_k\rangle_{n_k} = \sigma'_k$ or both states σ_k and σ'_k are $false$, provided no states in either derivation (apart from σ_k and σ'_k) are of the form $\langle A_1, S', B', T'\rangle_{n'}$ or $\langle A_2, S', B', T'\rangle_{n'}$ respectively.*

Proof
As with the proof of Lemma 6, it suffices to prove one direction of the "iff" only, since the other direction is symmetric. We prove the " $\Longrightarrow$ " direction by direct proof.

There are two cases to consider. The first is that the derivation is of zero length, i.e. $\sigma = \sigma_k$, then $\sigma'_k = \sigma'$ satisfies the hypothesis.

The second case is derivations of non-zero length. Let D_k be the derivation $\sigma \rightarrowtail \sigma_k$ above. We can write $D_k = D_{k-1} \rightarrowtail \sigma_k$, where the last state in D_k is $\sigma_{k-1} = \langle G +\!\!+ A_1, S_{k-1}, B_{k-1}, T_{k-1}\rangle_{n_{k-1}}$ for some non-empty G. By Lemma 5 there is a derivation from σ' to the state $\sigma'_{k-1} = \langle G +\!\!+ A_2, S_{k-1}, B_{k-1}, T_{k-1}\rangle_{n_{k-1}}$. Call this derivation D_{k-1}.

Consider the transition from σ_{k-1} to σ_k. We can apply exactly the same transition to σ'_{k-1} to derive σ'_k of the above form (using the same argument as in the proof of Lemma 6).

Therefore if $\sigma = \langle G +\!\!+ A_1, S, B, T\rangle_n \rightarrowtail^* \langle A_1, S_k, B_k, T_k\rangle_{n_k} = \sigma_k$ then for all A_2 we have that $\sigma' = \langle G +\!\!+ A_2, S, B, T\rangle_n \rightarrowtail^* \langle A_2, S_k, B_k, T_k\rangle_{n_k} = \sigma'_k$ provided the conditions noted in the Lemma above hold. By symmetry the other direction of the "iff" also holds. □

We are ready for the main result. First we give a formal definition of the confluence test.

Definition 23 (Confluence Test) *A program P passes our* confluence test *if*

1. *P is terminating;*

2. *All occurrences in P are matching complete or matching independent; and*
3. *All matching complete occurrences in P are order independent.*

Also, the implementation uses $trivial(S, c, B)$ as the wakeup policy.

We show that the test outlined above actually proves confluence. First we show that it at least proves *local confluence*, which is a weaker form of confluence.

Definition 24 (Local Confluence) *A CHR program is* local confluent *if the following holds for all states σ_0, σ_1 and σ_2 where σ_0 is a reachable state: If $\sigma_0 \rightarrowtail \sigma_1$ and $\sigma_0 \rightarrowtail \sigma_2$ then σ_1 and σ_2 are joinable with respect to σ_0.*

The only difference between local confluence and confluence is that states σ_1 and σ_2 are derived after a single transition step, rather than an arbitrary number of steps. We can now state the Lemma.

Lemma 7 (Local Confluence Test) *Let P be a CHR program that satisfies Definition 23, then P is locally confluent.*

Proof
Direct proof. We show that all reachable states σ such that if $\sigma \rightarrowtail_1 \sigma_1$ and $\sigma \rightarrowtail_2 \sigma_2$ then σ_1 and σ_2 are joinable. Note the notation $\rightarrowtail_1$ and $\rightarrowtail_2$ representing the transitions from σ to σ_1 and σ_2 respectively.

By inspection, all of the conditions for ω_r transitions are pairwise mutually exclusive. In other words, it is not possible that $\rightarrowtail_1$ and $\rightarrowtail_2$ are different transitions, thus $\rightarrowtail_2 = \rightarrowtail_1$.

Assume $\rightarrowtail_1$ and $\rightarrowtail_2$ are one of **Activate**, **Reactivate**, **Drop** or **Default**. By inspection, all of these transitions are deterministic, hence $\sigma_1 = \sigma_2$ thus the two states are trivially joinable.

The remaining cases for $\rightarrowtail_1$ (and $\rightarrowtail_2$) are as follows.

CASE **Solve**:

Then σ is of the form $\langle [c|A], S, B, T\rangle_n$, where c is a built-in constraint, and σ_1 and σ_2 are both of the form

$$\langle trivial(S, c, B) +\!\!+ A, S, c \wedge B, T\rangle_n = \langle A, S, c \wedge B, T\rangle_n$$

The other case is that $\sigma_1 = \sigma_2 = false$. Either way $\sigma_1 = \sigma_2$ and hence are trivially joinable.

CASE **Simplify**:

State σ is of the form $\langle [c\#i : j|A], S, B, T\rangle_n$ and there are two (possibly identical) matchings M_1 and M_2 which satisfy the conditions for **Simplify**. Then σ_1 and σ_2 are given by

$$\sigma_m = \langle goal(r_m) +\!\!+ A, S - delete(M_m), \theta(r_m, M_m) \wedge B, \{entry(r_m, M_m)\} \cup T\rangle_n$$

for $m = 1$ and $m = 2$ respectively. Here, r_1 and r_2 are two distinct renamings of the rule used in the transition.

There are two possible cases to consider for the occurrence j.

1. j is *matching complete*: By the definition of **Simplify** the active constraint $c\#i$ is deleted, thus $c\#i \in \mathit{delete}(M_1)$ and $c\#i \in \mathit{delete}(M_2)$. Thus the only way for occurrence j to be matching complete is that there is only one possible matching, i.e. $M_1 = M_2$. Then σ_1 and σ_2 must be variants and therefore are trivially joinable.
2. j is *matching independent* (and matching incomplete): Matching independence requires the states given by

$$\sigma'_m = \langle \mathit{goal}(r_m), S - \mathit{delete}(M_m), \theta(r_m, M_m) \wedge B, \{\mathit{entry}(r_m, M_m)\} \cup T\rangle_n$$

for $m = 1$ and $m = 2$ are joinable. This means that there exists variant states σ''_1 and σ''_2 such that $\sigma'_1 \rightarrowtail^* \sigma''_1$ and $\sigma'_2 \rightarrowtail^* \sigma''_2$. We write σ''_1 and σ''_2 as

$$\sigma''_m = \langle A''_m, S''_m, B''_m, T''_m\rangle_{n''_m}$$

for $m = 1$ and $m = 2$. Then by Lemma 6, we have that $\sigma_1 \rightarrowtail^* \sigma_3$ and $\sigma_2 \rightarrowtail^* \sigma_4$ where σ_3 and σ_4 are given by

$$\sigma_m = \langle A''_m ++ A, S''_m, B''_m, T''_m\rangle_{n''_m}$$

for $m = 3$ and $m = 4$. Clearly if σ''_1 and σ''_2 are variants then σ_3 and σ_4 are variants, therefore σ_1 and σ_2 are joinable.

CASE **Propagate**:

Matching independence is not applicable because **Propagate** does not delete the active constraint.

State σ is of the form $\langle [c\#i : j|A], S, B, T\rangle_n$ and there are two (possibly identical) matchings M_1 and M_2 which satisfy the conditions for **Propagate**. Then σ_1 and σ_2 are given by

$$\sigma_m = \langle \mathit{goal}(r_m) ++ [c\#i : j|A], S - \mathit{delete}(M_m), \\ \theta(r_m, M_m) \wedge B, \{\mathit{entry}(r_m, M_m)\} \cup T\rangle_n$$

for $m = 1$ and $m = 2$ respectively. Once again, r_1 and r_2 are two distinct renamings of the rule used in the transition.

Thanks to order independence, we know that the states given by

$$\sigma_{(m_1,m_2)} = \langle \mathit{goal}(r_{m_1}), S_{m_2} - \mathit{delete}(M_{m_1}), \theta(r_{m_1}, M_{m_1}) \wedge B_{m_2}, \\ \{\mathit{entry}(r_{m_1}, M_{m_1})\} \cup T_{m_2}\rangle_{n_{m_2}}$$

are joinable where S_{m_2}, B_{m_2}, T_{m_2} and n_{m_2} are given by final states arising from

$$\langle \mathit{goal}(r_{m_2}), S - \mathit{delete}(M_{m_2}), \theta(r_{m_2}, M_{m_2}) \wedge B, \{\mathit{entry}(r_{m_2}, M_{m_2})\} \cup T\rangle_n \rightarrowtail^* \\ \langle A_{m_2}, S_{m_2}, B_{m_2}, T_{m_2}\rangle_{n_{m_2}}$$

for $(m_1, m_2) = (1, 2)$ and $(m_1, m_2) = (2, 1)$. This means that there exists variant states σ''_1 and σ''_2 such that $\sigma_{(1,2)} \rightarrowtail^* \sigma''_1$ and $\sigma_{(2,1)} \rightarrowtail^* \sigma''_2$. We write σ''_1 and σ''_2 as

$$\sigma''_m = \langle A''_m, S''_m, B''_m, T''_m\rangle_{n''_m}$$

for $m = 1$ and $m = 2$.

Consider σ_1 defined above. Then by Lemma 1 we have that

$$\sigma_1 \rightarrowtail^* \langle [c\#i : j|A], S_1, B_1, T_1\rangle_{n_1} = \sigma'_1$$

By matching completeness $M_2 \subseteq S_1$. W.l.o.g. we can assume $\mathit{entry}(r_2, M_2) \notin T_1$ and the guard still holds (because the built-in store in monotonic by assumption). Thus

$$\sigma_1' \rightarrowtail_{\mathit{propagate}} \langle \mathit{goal}(r_2, M_2) +\!\!+ [c\#i : j|A], S_1 - \mathit{delete}(M_2), \\ B_1, \{\mathit{entry}(r_2, M_2)\} \cup T_1\rangle_{n_1}$$

By Lemma 6 we have that

$$\sigma_1 \rightarrowtail^* \sigma_1' \rightarrowtail^*= \langle A_1'' +\!\!+ [c\#i : j|A], S_1'', B_1'', T_1''\rangle_{n_1''}$$

We can apply a symmetric argument to similarly derive

$$\sigma_2 \rightarrowtail^* \langle A_2'' +\!\!+ [c\#i : j|A], S_2'', B_2'', T_2''\rangle_{n_2''}$$

These states must be variants because σ_1'' and σ_2'' (defined above) are also variants. Therefore σ_1 and σ_2 are joinable.

We have shown that if $\sigma \rightarrowtail \sigma_1$ and $\sigma \rightarrowtail \sigma_2$ then σ_1 and σ_2 are joinable. Therefore P is locally confluent. □

Finally, we can state the main result.

Theorem 3 (Confluence Test) *Let P be a CHR program that satisfies Definition 23, then P is confluent.*

Proof
By Lemma 7 program P is locally confluent. By definition P is terminating. Therefore by Newman's Lemma (Newman 1942) P is confluent. □

6 Implementation of Confluence Test

So far we have introduced some conditions, e.g. matching completeness etc., and shown that if these conditions hold for a given program P, then P is confluent. The confluence test is undecidable in general, since it relies on termination, however in this section we discuss how a modern CHR compiler can test (with some assumptions) if these conditions hold based on information it collects from CHR program analysis (Schrijvers et al. 2005). We allow the tests to be inaccurate, in that it is allowed to reject programs that are confluent, but not the other way around. Later in this paper we try the confluence tests on several examples.

The tests outlined below have been implemented as part of the Mercury CHR compiler, which we will refer to as *confluence checker* from now on. The confluence checker implements partial tests for fixedness of CHR constraints, matching completeness and matching independence, and relies on user annotation for determining order independence except for a few cases discussed below. The confluence checker assumes termination, which (as usual) is left to the programmer to decide.

The first part of the confluence test the Mercury CHR compiler tests for is groundness/fixedness, since this is required for the usage of the trivial wakeup policy to be correct. The Mercury compiler already has access to this information, since the user must write a `mode` declaration for each CHR constraint. If the modes for every argument for each CHR constraint are '`in`', then the program passes this part of the confluence test.

In Mercury CHR, constraints are allowed to have mode '`out`' under restricted conditions. Let c be a CHR constraint with an '`out`' argument represented by v (which must be a '`new`' variable at runtime). A constraint `p`/n is *never-stored* if it never exists in the CHR store S when it is not the top of the activation stack (see (**?**; Holzbaur et al. 2005) for a more complete discussion of never-stored). A constraint with mode '`out`' is acceptable if:

1. c is *never-stored* anywhere in the program ;
2. all possible rule bodies called by c (by firing a rule) either bind v to a ground value or fail.

The never-stored requirement will ensures that c is never in the CHR store whenever a built-in constraint is **Solve**d, hence we avoid the nondeterminism.

Example 20 *The* `find(A,X)` *constraint in Example 2 is a classic example of a constraint with a argument with mode '*`out`*'. Its mode declaration provided by the programmer is as follows.*

```
:- mode find(in,out) is semidet.
```

When called, variable `X` *will be unbound, but will be bound to the representative of the equivalence class for* `A` *at the end of the execution of the goal.*

Since constraints with '`out`' modes can never be woken up, the usage of the trivial wakeup policy is still correct. Therefore, such programs also pass the confluence checker.

The confluence checker also uses information about never-stored and functional dependencies (Duck and Schrijvers 2005) to determine how many possible matchings there are for each occurrence in a given rule. Functional dependencies for CHR constraints define characteristics of the constraint store. For example the functional dependency: $entry(K, V) :: \{K\} \rightsquigarrow \{V\}$ states that for all `entry`/2 constraints in the store, the key K *functionally determines* the value V, that is for each K there is at most one V where $entry(K, V)$ is in the store. We can extend the notion of functional determination as follows. Let F be a set of functional dependencies that hold, then $close_F(V)$ be the smallest set such that $close_F(V) \supseteq V$ and for each $p(\bar{X}) :: V_0 \rightsquigarrow V_1$ in F, if $V_0 \subseteq close_F(V)$ then $V_1 \subseteq close_F(V)$. We say a constraint c *functionally determines* a set of constraints C if F is the set of functional dependencies that hold for each constraint in $c \cup C$, and $vars(C) \subseteq close_F(vars(c))$. Clearly if an active constraint c functionally determines the head of the rule, then there can be at most one matching.

For occurences where there are zero or one possible matchings, then the occurrence is trivially matching complete. This is very common in many programs. Otherwise if there are multiple possible matchings, the confluence checker then checks for matching completeness as follows. Suppose that there are at least two matchings M_1 and M_2 for a given occurrence, then for matching completeness there are are two cases to consider: applying the **Simplify** or **Propagate** transition on M_1 *directly* deletes a constraint $c\#i \in M_2$ (e.g. Example 11); or executing the rule body *indirectly* deletes $c\#i \in M_2$ (e.g. Example 17).

For the first part, we check for direct deletion as follows. Let $(H_1 \backslash H_2)$ be the head of the rule, and let c be the active constraint, then there are two cases to consider. The first case is when the active constraint is deleted by the occurrence (i.e. a member of H_2). To be matching complete, it must be that there can only ever be zero or one possible matchings (since c must be present in all matchings). To check this we use never-stored and functional dependency information to check if either one of $chr(H_1) \cup chr(H_2) - \{c\}$ is never-stored (therefore there cannot be any matchings) or the head $chr(H_1) \cup chr(H_2)$ is functionally determined by c (therefore there can only be one possible matching).

The other case where active c is not deleted by the occurrence is more complicated. We allow for four possible sub-cases:

1. the rule is a propagation rule, i.e. $H_2 = \emptyset$;
2. one of $chr(H_1) \cup chr(H_2) - \{c\}$ is never-stored;
3. the active constraint c functionally determines $chr(H_1) \cup chr(H_2)$;
4. for all $d \in chr(H_2)$ we have that d functionally determines $chr(H_1) \cup chr(H_2) - \{c\}$ and for all $d_1, d_2 \in (chr(H_1) \cup chr(H_2) - \{c\})$ we have that the predicate symbols of d_1 and d_2 are distinct.

Propagation rules can never directly delete constraints from any matching by definition, so they are safe. The second case and third cases are also trivially safe, since they imply there is only ever zero or one possible matching. The fourth case is more complicated. Suppose that for active $c\#i$ there is a constraint $d\#j$ and two matchings M_1 and M_2 such that $d\#j \in delete(M_1)$ and $d\#j \in delete(M_2)$. Then it must be that $M_1 - \{c\#i\} = M_2 - \{c\#i\}$, hence $M_1 = M_2$, otherwise d does not functionally determine the matching. Therefore, for all matchings M_1 and M_2 it must be that $delete(M_1) \cap delete(M_2) = \emptyset$.

Example 21 *Consider the following rule with three constraints in the head, and assume that all of these constraints have set semantics (at most one copy of each constraint can be in the store at any time).*

```
p \ q, r(X) <=> true.
```

Firstly note that the body cannot indirectly delete any constraint from any matching. The occurrence for `r(X)` is matching complete because `p` and `q` are both (trivially) functionally determined by the active constraint (thanks to set semantics).

The occurrence of `p` is not matching complete because neither `p` nor `q` functionally determine `r(X)`. However, if we were to modify the rule to the following, then the occurrence of `p` is matching complete.

```
p \ q(X), r(X) <=> true.
```

Now both `q(X)` and `r(X)` functionally determine each other.

The second part of the matching completeness check tests if the body can indirectly delete a constraint from another matching. This information can be read from a call-graph of CHR constraints, and by examining the heads of rules to determine which CHR constraints can delete other CHR constraints.

Example 22 *For example, consider the program from Example 17.*

```
r1 @ p, q(X) ==> r(X).
r2 @ p, r(a) <=> true.
```

The call graph reveals that the body of rule `r1` *calls constraints of predicate symbol* `r`*/1. By examining the heads of the rules, we see that an active* `r`*/1 constraint may delete a* `p` *constraint. Therefore calling the body of rule* `r1` *may delete the active* `p` *constraint, hence the matching completeness check must fail.*

If an occurrence fails matching completeness, then the confluence checker will try and prove matching independence, i.e. the choice of matching does not matter. Recall that matching independence is only applicable to occurrences where the active constraint is the only constraint deleted by the rule. A very simple matching independence test is to check if the free variables in the rule body are contained in the free variables of the active constraint. This was trivially true in Example 18, where the set of free variables in the body is empty. We can improve the matching completeness check by also allowing variables that are functionally determined by the active constraint.

Example 23 *Consider the following rule.*

```
r(Z), q(X,Y) \ p(X) <=> t(X,Y).
```

The occurrence for `p(X)` *is not matching independent because variable* `Y` *appears in the body, but not in the active constraint. If however there exists a functional dependency* $q(X, Y) :: \{X\} \rightsquigarrow \{X, Y\}$ *then the occurrence is matching independent.*

Currently the Mercury confluence checker assumes all occurrences are order independent by default, however the programmer can turn on order independence checking via a flag to the compiler. The order independence check is currently very weak, it involves finding all occurrences with more than one possible matching, and then checking if the rule body contains only built-in constraints,[12] or if the rule body is functionally determined by the active constraint. The programmer can also declare certain constraints and/or rules as "order independent", which narrows the checking to potential problem areas.

The order independence test could be improved by automatically checking some common CHR programming idioms. One such idiom is using constraints to accumulate some value, as was the case in Example 19. This can be generalised as follows. Suppose we have a rule of the form:

[12] This is a surprisingly common case, since many rules simply have '`true`' as the body.

`p(`X`), p(`Y`) <=> p(`X *op* Y`).`

Where *op* is some binary operator that is commutative and associative, e.g. addition $X+Y$ or set union $X \cup Y$, etc., then it does not matter which order goals of the form $\texttt{p}(X_1), ..., \texttt{p}(X_n)$ are called. Therefore any rule body consisting of a $\texttt{p}(X)$ constraint is order independent.

6.1 Experimental Results: Confluence Test

This section investigates the confluence of four CHR programs using our confluence checker. The programs are:

- `union` – Example 2 (see (Schrijvers and Frühwirth 2004)); and
- `ray` – a simple ray tracer;
- `bounds` – an extensible bounds propagation solver;
- `dijkstra` – Dijkstra's shortest path algorithm implemented in CHRs (see (Sneyers et al. 2006))

These programs are chosen because either they were

- implemented before the confluence test and checker were invented; or
- implemented by those who are not authors of this paper.

These conditions are designed to minimize the possibility of knowledge of the confluence test influencing how the program is implemented. We believe that knowledge of the confluence test encourages a more "deterministic" style of CHR programming, which is more likely to pass.

Each program is tested twice: the `.orig` version is the original program without any modifications (apart from being ported to Mercury CHR). The `.fixed` version is the same as `.orig` except (1) any bug detected by the confluence test is fixed; and (2) an appropriate rule is added for each implicit never-stored constraint or constraints with functional dependencies. For example, a constraint $\texttt{p}/n$ is implicitly never-stored if it is always deleted before it is stored (considering late storage), but the compiler's analysis is too weak to detect this. In such cases, the `.fixed` version of the program includes an explicit rule

```
p(_,...,_) <=> error("not never-stored").
```

which the compiler understands as enforcing the never-stored condition. The compiler can now detect more never-stored constraints, and take this information into account when deciding confluence. Similarly for implicit functional dependencies.

The results of our experiments are shown in Figure 4. The column $\neg MC/I$ lists the number of occurrences in the program that are not matching complete nor matching independent. The column $\neg OI$ lists the number of occurrences that are not order independent. Both indicate potential sources of non-confluence, and are reported as warnings by the Mercury CHR compiler.

Overall, the `.fixed` versions of the program – with bug fixes and explicity never-stored/functional-dependency rules – have fewer occurrences that report non-confluence warnings. Most of the improvement is in the number of occurrences $\neg MC/I$.

Prog.	rules	constraints	occurrences	$\neg MC/I$	$\neg OI$
`union.orig`	14	11	15	5	1
`union.fixed`	18	11	21	0 (-5)	0 (-1)
`ray.orig`	32	19	52	1	3
`ray.fixed`	33	19	54	0 (-1)	3 (=)
`bounds.orig`	79	38	123	4	3
`bounds.fixed`	79	38	123	3 (-1)	3 (=)
`dijkstra.orig`	28	16	42	9	3
`dijkstra.fixed`	30	16	46	3 (-6)	3 (=)

Fig. 4. Summary of results from the confluence test.

The naive union find algorithm is shown in Example 2. The confluence checker finds 5 matching completeness problems associated with 3 rules. Each of these problems exposed implicit assumptions about functional dependencies and never-stored constraints.

For example, the confluence checker complains that the following rule is not matching complete for all occurrences.

```
link @ link(A,B), root(A), root(B) <=> arrow(B,A), root(A).
```

This is solved by the addition of two rules: the first declares `root`/1 to have set semantics and the second explicitly deletes unused `link`/2 constraints (the programmer was implicitly assuming that `link`/2 will always fire this rule, however the confluence checker cannot detect this). The revised version is

```
root(A) \ root(A) <=> true.
link @ link(A,B), root(A), root(B) <=> arrow(B,A), root(A).
link(_,_) <=> true.
```

which passes the confluence checker.

The occurrence for `find`/2 in the `findRoot` rule is also reported as matching incomplete.

```
findRoot @ root(A) \ find(A,X) <=> X = A.
```

This is because the programmer intends there to be a single `root(`A`)` constraint per A, however the confluence checker cannot detect this. The problem is resolved in `union.fixed` by adding the following rule.

```
root(A) \ root(A) <=> true.
```

Similarly, the `find`/2 occurrence in the `findNode` rule is also matching incomplete.

```
findNode @ arrow(A,B) \ find(A,X) <=> find(B,X).
```

In this case the confluence checker does not know that there is one `arrow(`A`,`B`)` constraint per A. The problem is resolved in `union.fixed` by the following rule.

```
arrow(A,_) \ arrow(A,_) <=> true.
```

In the `ray` program the matching completeness problem appeared in:

```
lr1 @ intersection(IP,Id,D) \ light(LP,C) <=>
                                             light_ray(LP,IP,C,Id).
```

The `intersection` constraint is in fact an accumulator which keeps track of the nearest intersection with an object and the ray from the eye point.

```
int_near @ intersection(_,_,D1) \ intersection(_,_,D2) <=>
                                                  D1 =< D2 | true.
```

This rule ensures that there is at most one possible `intersection` constraint in the store at once, however the functional dependency analysis in the compiler is too weak to detect this (because it currently does not take the guard into account). This is fixed in `ray.fixed` by the user asserting the functional dependency to the compiler, which currently is managed by adding the following rule.

```
int_fd @intersection(_,_,_) \ intersection(_,_,_) <=> true.
```

Almost all of the confluence problems in the other programs were fixed using a similar method. The exception is `bounds.orig`, where the confluence analysis detects a bug. The following rule is not matching complete nor independent when `kill(Id)` is active since there are (potentially) many possible matchings for the `delayed_goals` partner.

```
kill @ kill(Id), delayed_goals(Id,X,_,...,_) <=> true.
```

Here `delayed_goals(Id,X,_,...,_)` represents the delayed goals for bounds solver variable X. The code should be

```
kill1 @ kill(Id) \ delayed_goals(Id,X,_,...,_) <=> true.
kill2 @ kill(_) <=> true.
```

This highlights how a simple confluence analysis can be used to discover bugs.

Order independence is harder to detect, and there is little the programmer can do to remove the related warning messages (other than disable them). The once exception is in the `union.orig` program, where adding a explicit never-stored reduced the number of occurrences the confluence checker considers. This resulted in the offending occurrence being removed from consideration, and hence no warning.

Dispite fixing bugs and adding explicit rules, some of the confluence warnings are not removed. For example, the confluence analysis reports warnings for the rules for bounds propagation themselves, e.g. the following rule handles bounds propagation for a `leq`/2 constraint.

```
leq @ leq(X,Y), bounds(X,LX,UX), bounds(Y,LY,UY) ==>
                                 bounds(X,LX,UY), bounds(Y,LX,UY).
```

The problem is that the constraint `bounds(`X`,`L`,`U`)` which stores the lower L and upper U bounds of variable X has complex self-interaction. Two `bounds` constraints for the same variable can interact using, for example,

```
b2b @ bounds(X,L1,U1), bounds(X,L2,U2) <=>
                          bounds(X,max(L1,L2),min(U1,U2)).
```

Imagine an active bounds constraint visiting one of the occurrences for rule `leq`. The body of `leq` calls a new `bounds` which may delete the active constraint, and therefore the occurrences are indeed not matching complete.

Although the program contains rules which are not matching complete, in this case the matching incompleteness does not indicate a bug (the rules are still confluent). In this case confluence can be established by showing that the propagation rules, together with the other rules for the `bounds` constraint, are confluence under the theoretical semantics. Confluence under the refined semantics then follows because of Corollary 1. Unfortunately, proving confluence under the theoretical semantics is beyond the current implementation, so this is left for the programmer. A similar argument can be made for all of the remaining matching incompleteness problems reported by the confluence checker.

7 Related Work

The presentation of the theoretical operational semantics of CHRs differs from others that have appeared in past literature, e.g. in (Frühwirth 1998; Abdennadher 1997), in several ways. In this section we argue that our formalisation subsumes the previous versions.

The main difference is the interpretation of simpagation rules. Previous versions of the operational semantics treated simpagation rules as shorthand for simplification rules. Specifically, a simpagation rule of the form

$$h_1, \ldots, h_l \backslash h_{l+1}, \ldots, h_n \Longleftrightarrow g \mid b_1, \ldots, b_m$$

is treated as a simplification rule of the form

$$h_1, \ldots, h_l, h_{l+1}, \ldots, h_n \Longleftrightarrow g \mid h_1, \ldots, h_l, b_1, \ldots, b_m$$

This translation does not necessarily preserve operational equivalence under (our version of) ω_t, since the copies of $h_1, \ldots, h_l$ will be assigned new constraint numbers when they are (re)executed in the body. This may effect the behaviour of the propagation history.

The new interpretation of simpagation rules effectively extends the operational semantics in (Frühwirth 1998; Abdennadher 1997) (the old interpretation can be emulated under ω_t by translating simpagation rules explicitly). It should be noted that some theoretical results for CHRs, e.g. Abdennadher's confluence test (Abdennadher 1997), may depend on the old interpretation of simpagation rules.

Most of the other differences are trivial. For example, we represent the constraint store as a (multi)set of $c\#i$ tuples where each i is unique. In the formalisation presented in (Abdennadher 1997), the constraint store is represented as a conjunction of constraints, where repeats in the conjunction are allowed (i.e. assuming non-idempotent conjunction). Both formalisations of the constraint store are isomorphic.

Propagation histories are also handled differently. In our approach, entries in the propagation history are tuples of constraint numbers (and the rule token), whereas in other versions of the operational semantics (e.g. (Abdennadher 1997)), entries record the CHR constraints themselves. This is implemented as follows. When a new constraint c is introduced into the store (via the **Introduce** transition), a so-called *token set*[13] is generated for constraint c. Each token is of the form $r@H'$, where $(r\ @\ H \implies g \mid B)$ is a propagation rule in P (r is the rule identifier), and H' is a conjunction of constraints in $c \wedge S$ and c is a conjunct in H'. When propagation rule r fires on constraints $H' \subseteq S$, a corresponding token $r@H'$ must be present in the propagation history, and is removed when the rule fires. Notice this is the the opposite approach, removing elements, rather than adding elements to propagation history as the derivation progresses.

Example 24 *For example, consider the following propagation rule.*

```
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).
```

Under our representation, firing the propagation rule on the following matching constraints, $\texttt{leq}(A,B)\#1, \texttt{leq}(B,C)\#2$*, adds the entry* $[1,2,\texttt{transitivity}]$ *to the propagation history.*

Under the alternative formalisation, a token $\texttt{transitivity}@(\texttt{leq}(A,B) \wedge \texttt{leq}(B,C))$ *is present in the propagation history and removed when the rule fires.*

These approaches are isomorphic, that is the operational semantics are equivalent no matter which formalisation of the propagation history is chosen, however our approach more closely mimics what most CHRs systems actually implement.

8 Conclusion

In this paper we have formalized the refined operational semantics of CHRs, which are a popular semantics used by almost all current CHR implementations that we are aware of. The refined operational semantics define a powerful and expressive language, where simple database operations and fixed-point computations are straightforward to implement.

We have proved several important results, such as correctness, soundness, completeness, termination and confluence. The correctness results state that every derivation in the refined semantics map to a derivation under the theoretical semantics with the same answer. Also, every reachable final state in the refined semantics maps to a reachable final state in the theoretical semantics, therefore the refined semantics correctly implement the theoretical semantics. The other results including soundness, completeness, termination and confluence follow from correctness. Soundness and completeness are important results from a theoretical point of view.

Termination is an essential property of any program, including CHR programs. The termination result ensures that a termination proof in the theoretical semantics

[13] Although it is called the token "set", it is really a multiset of tokens (repeats are allowed).

immediately applies to the refined semantics. This is useful since termination of CHRs has been looked at in (Frühwirth 1999), and therefore these results carry to the refined semantics. It may be easier to prove termination under the refined semantics because of the increased determinism, hence there are less derivations to consider, however it is still left to the programmer.

The refined operational semantics for Constraint Handling Rules provides a powerful and expressive language, ideal for applications such as compilers, since fixpoint computations and simple database operations are straightforward to program. The disadvantage of CHRs over other possible languages is that CHRs do not have a fully deterministic operational semantics, so to counter this problem the programmer usually aims to write confluent CHRs programs. Unfortunately, the Abdennadher confluence test (Abdennadher 1997) is too strong for the refined operational semantics, so we have presented a novel static confluence checker based on information obtained from standard CHR program analysis.

The confluence test identifies four properties that if satisfied, guarantees confluence under the refined semantics. These are: termination, trivial wakeup policy, matching completeness or independence, and order independence. The termination requirement is solely left for the programmer, and order independence typically requires help from the programmer (in current implementations). Groundness and matching completeness/independence can be checked automatically in modern CHR compilers.

We implemented a confluence checker for Mercury CHRs based on the confluence test, and evaluated the checker on four CHR programs: a union-find algorithmg, a simple ray tracer, a bounds propagation solver, and a shortest path program. By far testing for matching completeness or independence is the most useful, since the majority of all occurrences the case studies were either matching complete, matching independent, or in one instance indicated a bug. The exceptions occur when the programmer is relying on confluence under the theoretical semantics, as with some of the rules in the `bounds` example.

Matching completeness exposes the programmer's implicit assumptions about functional dependencies, as was shown by the case studies. This is good because it encourages the programmer to declare functional dependencies explicitly, which has other benefits, such as optimisation.

Order independence remains a difficult property to test for, hence the confluence checker usually requires help in the form of user annotations. In order not to overwhelm the programmer, the compiler can try its best to exclude as many occurrences as possible, for example, when the body of the rule contains only built-in constraints, and when the body is functionally determined by the active constraint. A more sophisticated compiler can also try to exclude some other cases, e.g., when the body of a rule is calling constraints that just accumulate some value, etc.

Unfortunately the confluence test performs poorly when the programmer writes code with complex interactions, e.g. the `bounds` constraint and propagators. In these cases the programmer is usually relying on confluence under the theoretical operational semantics, which our current implementation cannot detect (although this may be future work). In such cases the programmer can ignore, or disable the confluence checker, or reformulate the program so that it complies with the confluence test in Definition 23.

References

Abdennadher, S. 1997. Operational semantics and confluence of constraint propagation rules. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, G. Smolka, Ed. LNCS 1330. Springer-Verlag, 252–266.

Abdennadher, S., Frühwirth, T., and Muess, H. 1999. Confluence and Semantics of Constraint Simplification Rules. *Constraints 4,* 2, 133–166.

Duck, G. and Schrijvers, T. 2005. Accurate functional dependency analysis for constraint handling rules. In *Proceedings of the 2nd Workshop on Constraint Handling Rules*, T. Schrijvers and T. Frühwirth, Eds. 109–124.

Frühwirth, T. 1998. Theory and practice of constraint handling rules. *Journal of Logic Programming 37*, 95–138.

Frühwirth, T. 1999. Proving termination of constraint solver programs. In *New Trends in Contraints, Joint ERCIM/Compulog Net Workshop*. LNCS 1865. Springer-Verlag, 298–317.

Frühwirth, T. 2005. Parallelizing union-find in constraint handling rules using confluence. In *Proceedings of the International Conference on Logic Programming*, M. Gabbrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. Springer, 113–127.

Holzbaur, C., de la Banda, M. G., Stuckey, P., and Duck, G. 2005. Optimizing compilation of constraint handling rules in HAL. *Theory and Practice of Logic Programming 5,* 4–5, 503–532.

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

Holzbaur, C. and Frühwirth, T. 2000. A Prolog constraint handling rules compiler and runtime system. *Journal of Applied Artificial Intelligence 14,* 4.

Holzbaur, C., Stuckey, P., de la Banda, M. G., and Jeffery, D. 2001. Optimizing compilation of constraint handling rules. In *Logic Programming: Proceedings of the 17th International Conference*, P. Codognet, Ed. LNCS 2237. Springer-Verlag, 74–89.

Newman, M. 1942. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics 43,* 2, 223–243.

Schrijvers, T. 2005. http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/.

Schrijvers, T. and Frühwirth, T. 2004. Implementing and Analysing Union-Find in CHR. Tech. Rep. CW 389, K.U.Leuven, Department of Computer Science. July.

Schrijvers, T. and Frühwirth, T. 2006. Optimal union-find in constraint handling rules. *Theory and Practice of Logic Programmin 6,* 1&2, 213–224.

Schrijvers, T., Stuckey, P., and Duck, G. 2005. Abstract interpretation for constraint handling rules. In *PPDP'05: Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, A. Felty, Ed. ACM, Lisbon, Portugal, 218–229.

Sneyers, J., Schrijvers, T., and Demoen, B. 2006. Dijkstra's algorithm with fibonacci heaps: an executable description in CHR. In *Proceedings of the 20th Workshop on Logic Programming*, M. Fink, H. Tompits, and S. Woltran, Eds. 182–191.