The Refined Operational Semantics of Constraint Handling Rules

Gregory J. Duck\textsuperscript{1}, Peter J. Stuckey\textsuperscript{1}, María García de la Banda\textsuperscript{2}, and Christian Holzbaur\textsuperscript{3}

\textsuperscript{1} Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
\{gjd,pjs\}@cs.mu.oz.au
\textsuperscript{2} School of Computer Science and Software Engineering
Monash University, Vic. 3800, Australia
mbanda@csse.monash.edu.au
\textsuperscript{3} holzbaur@chello.at

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 \textit{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.

1 Introduction

Constraint Handling Rules (CHRs) are a high-level rule-based programming language commonly used to write constraint solvers. The theoretical operational semantics of CHRs is relatively high level with several choices, such as the order in which transitions are applied, left open. Therefore, only confluent CHR programs, where every possible execution results in the same result, have a guaranteed behaviour.

This paper looks at the \textit{refined} operational semantics, a more specific operational semantics which has been implicitly described in \cite{10,11}, and is used by every Prolog implementation of CHRs we know of. Although some choices are still left open in the refined operational semantics, both the order in which transitions are applied and the order in which occurrences are visited, is decided. Unsurprisingly, the decisions follow Prolog style and maximise efficiency of execution. The remaining choices, which matching partner constraints are tried first, and the order of evaluation of CHR constraints awoken by changes in variables they involve, are left as choices for two reasons. First it is very difficult to
see how a CHR programmer will be able to understand a fixed strategy in these cases. And second implementing a fixed strategy will restrict the implementation to be less efficient, for example by disallowing hashing index structures.

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 \((\text{gcd})\) between two integers using Euclid’s algorithm:

\[
\text{gcd1} @ \text{gcd}(0) \iff \text{true}.
\]
\[
\text{gcd2} @ \text{gcd}(N) \setminus \text{gcd}(M) \iff M >= N \mid \text{gcd}(M-N).
\]

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

This program, which appears on the CHR webpage [6], is non-terminating under the theoretical operational semantics. Consider the constraint store \(\text{gcd}(3), \text{gcd}(0)\). If the first rule fires, we are left with \(\text{gcd}(3)\) and the program terminates. If, instead, the second rule fires (which is perfectly possible in the theoretical semantics), \(\text{gcd}(3)\) will be replaced with \(\text{gcd}(3-0) = \text{gcd}(3)\), 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 [7] 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 \(\text{gcd}\) example above. Of course, it could also have worked against it, since swapping the order of the rules would lead to non-termination.

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

**Example 2.** Consider a CHR implementation of a simple database:

\[
11 @ \text{entry}(\text{Key}, \text{Val}) \setminus \text{lookup}(\text{Key}, \text{ValOut}) \iff \text{ValOut} = \text{Val}.
\]
\[
12 @ \text{lookup}(\_\_, \_\_) \iff \text{fail}.
\]

where the constraint \text{lookup} represents the basic database operations of key lookup, and \text{entry} represents a piece of data currently in the database (an entry in the database). Rule 11 looks for the matching entry to a lookup query and returns in \text{ValOut} the stored value. Rule 12 causes a lookup to fail if there is no matching entry. Clearly the rules are non-confluent in the theoretical operational semantics, since they rely on rule ordering to give the intended behaviour.

\footnote{Unlike Prolog, we assume the expression “\(m - n\)” is automatically evaluated.}