Optimizing Compilation and Computational Complexity of Constraint Handling Rules
Ph.D. thesis summary

Jon Sneyers

K.U. Leuven, Belgium
jon.sneyers@cs.kuleuven.be

1 Introduction

Constraint Handling Rules [1, 2] is a high-level programming language extension based on multi-headed committed-choice multiset rewrite rules. It can be used as a stand-alone language or as an extension to an existing host language. CHR systems have been implemented for nearly every Prolog system, and there are also CHR systems for Haskell, Java and C.

In the past few years there has been quite some research interest in improving the performance of CHR systems. This has lead to the introduction of several novel compiler optimizations aimed at improving the computational complexity (mostly time complexity) of CHR programs. Recently at least four Ph.D. theses focussed on this topic: Tom Schrijvers [3] and Gregory Duck [4] worked mainly on optimizing compilation, while Leslie De Koninck[1] [5] worked on computational complexity and compilation techniques for variants of CHR that deviate from the de facto standard refined operational semantics ω_r [6]. This paper gives a brief overview of the main results of the Ph.D. thesis of Jon Sneyers [7].

2 Optimizing Compilation

The first part of the thesis [7] introduces CHR and gives an overview of the state-of-the-art in optimizing CHR compilation. In the second part, a number of novel compiler optimizations are introduced. They are briefly discussed in this section. Most of these optimizations are necessary for achieving the main complexity results (discussed in part three of the thesis and Section 3 of this summary). They are implemented in the Leuven CHR system [3] in hProlog [8].

Guard reasoning. The abstract operational semantics ω_t of CHR is very nondeterministic. For example, the order in which rules are applied is not specified at all. Instantiations of the abstract operational semantics — for example the refined operational semantics — remove sources of nondeterminism and in that sense they give more execution control to the programmer. Rule guards may be redundant under a more instantiated semantics while being necessary in the

[1] A summary of Leslie's Ph.D. thesis can be found elsewhere in this volume.

abstract semantics. Expert CHR programmers tend to remove such redundant guards. Although this improves performance, it also destroys the local character of the logical reading of CHR rules: in order to understand the meaning of a rule, the entire program and the details of the instantiated operational semantics have to be taken into account. As a solution, we propose compiler optimizations that automatically detect and remove redundant guards.

Memory reuse. Repeatedly replacing a constraint with a new one is a typical pattern that, in current CHR implementations, does not have the space complexity one might expect. The extra space can be reclaimed using (host language) garbage collection, but this comes at a cost in execution time. Indeed, CHR programmers often see that more than half of the total runtime is spent on garbage collection. We therefore introduce two compiler optimizations, *in-place update* and *suspension reuse*, that drastically reduce the memory footprint of CHR programs. Both optimizations reuse suspension terms, the internal representation of CHR constraints, and avoid redundant indexing operations. The optimizations are defined formally and their correctness is proved. They were implemented and significant memory savings and speedups were measured.

Join ordering. A crucial aspect of CHR compilation is finding matching rules efficiently. Given an active constraint, searching for matching partner constraints corresponds to joining relations — a well-studied topic in the context of databases. The performance of join methods is determined by the efficiency of the indexing techniques and by join ordering. In the refined operational semantics ω_r, a different join ordering can be used for each active occurrence in a rule. Given a CHR program $\mathcal{P}$ and one of its active occurrences a, a join ordering strategy $\prec$ imposes a total order $\prec_a^{\mathcal{P}}$ on the partner constraints of a. We formulate a generic cost model to evaluate join orderings and we propose static and dynamic heuristics to implement a join ordering strategy.

3 Computational Complexity

As a stand-alone programming language CHR is Turing-complete. In fact, several subclasses of CHR are already Turing-complete. These computability properties of CHR are discussed in the beginning of part three of the thesis [7]. The rest of part three is mostly devoted to the computational complexity of CHR [9].

In order to investigate the computational complexity of CHR, we have introduced abstract CHR machines. These machines essentially execute one CHR rule (or more exactly, one ω_t transition) in every step. We define the time complexity of a CHR machine to be the number of steps it takes. This is unrealistic since finding an applicable CHR rule takes more than constant time in general. We thus have to investigate the relation between CHR machines and more realistic models of computation, in particular the RAM machine.

We now state the main results. Because of space limitations we have to refer to [7] for the underlying definitions, lemmas, and proofs. We just give the definitions of determined partner constraints and the dependency rank of a constraint occurrence, because they are central to the formulation of the theorems.

Definition 1 (determined partner). *Given a join ordering strategy $\prec$, a CHR program $\mathcal{P}$, and a set of valid goals $\mathcal{VG}$, we say an occurrence c is* determined *by the j-th occurrence of constraint a iff for all execution states σ that occur in a derivation $d \in \Delta^{\mathcal{H}}_{\omega_r}|_{\mathbb{G}}$ for some valid goal $\mathbb{G} \in \mathcal{VG}$, the following holds: if σ is of the form $\langle [a\#i:j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ (that is, the occurrence subprocedure for the j-th occurrence of constraint a is about to be executed), then a set semantic functional dependency for c holds in state σ, where the key arguments of c are fixed by a and all partners x for which $x \prec^{\mathcal{P}}_a c$.*

In other words, a partner constraint c is determined by a given (active) constraint occurrence of a if the following holds: whenever the partner constraint c is looked up, there is at most one match that needs to be considered.

Definition 2 (dependency rank). *The dependency rank of an (active) occurrence a is the number of non-determined partner constraints of a.*

Complexity meta-theorem. The dependency rank of an occurrence corresponds to the "real" nesting depth of the lookup iterations. Given a constraint store of size S, the worst-case complexity of searching for matching partner constraints of an occurrence with dependency rank d is $O(S^d)$. This observation leads to the following complexity meta-theorems, which improve upon earlier results [10]:

Theorem 1. *Given a CHR program $\mathcal{P}$ and a ω_t derivation d of length T which has a corresponding ω_r derivation, for which the maximal store size is S, m is the maximum dependency rank of the active occurrences in $\mathcal{P}$, and p is the number of propagation rule applications in d; assuming the host language constraints used in the guards and bodies of the rules of $\mathcal{P}$ can be evaluated in constant time; the Leuven CHR system compiles $\mathcal{P}$ to hProlog code which has, for the given derivation d, a time complexity $O(TS^{m+1})$ and a space complexity $O(S+p)$.*

Theorem 2. *If in the previous theorem, the CHR program is ground (i.e. all constraint arguments are ground), then $O(TS^m)$ time complexity can be achieved.*

Complexity-wise completeness. Now we show that "everything can be done in CHR with the right complexity". Given an arbitrary RAM machine program, we can simulate it in CHR using the simulator program RAMSIMUL (Fig. 1). The program takes $O(T+S)$ rule applications to simulate a RAM-machine with time complexity T and space complexity S. Using the above complexity meta-theorem we then show the following theorem:

Theorem 3. *An ω_t derivation for the program* RAMSIMUL*, with T steps and maximal store size S, can be executed in $O(T)$ time and $O(S)$ space.*

Proof. Follows from Theorem 2: the program RAMSIMUL is ground, it has no propagation rules so $p = 0$, and there is a join ordering strategy such that the maximal dependency rank $m = 0$.

```
i(L,init,A), m(A,B), maxm(M) \ c(L) <=> initm(M+1,B,L).
      initm(A,B,L) <=> A =< B | m(A,0), initm(A+1,B,L).
      initm(A,B,L), m(B,X) <=> A > B | m(B,0), maxm(B), c(L+1).
i(L,cnst,B,A) \ m(A,X), c(L) <=> m(A,B), c(L+1).
i(L,add,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X+Y), c(L+1).
i(L,sub,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X-Y), c(L+1).
i(L,mul,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X*Y), c(L+1).
i(L,div,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X//Y), c(L+1).
i(L,mov,B,A), m(B,Y) \ m(A,_), c(L) <=> m(A,Y), c(L+1).
i(L,imv,B,A), m(B,C), m(C,Y) \ m(A,_), c(L) <=> m(A,Y), c(L+1).
i(L,mvi,B,A), m(B,Y), m(A,C) \ m(C,_), c(L) <=> m(C,Y), c(L+1).
i(L,jmp,A) \ c(L) <=> c(A).
i(L,cjmp,A,J), m(A,0) \ c(L) <=> c(J).
i(L,cjmp,A,J), m(A,X) \ c(L) <=> X =\= 0 | c(L+1).
i(L,halt) \ c(L) <=> true.
```

Fig. 1. RAMSIMUL: Simulator of standard RAM machines

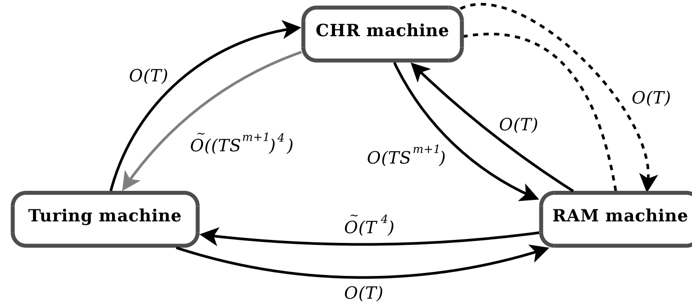

Fig. 2. Time complexity relationships between Turing, RAM, and CHR machines

Figure 2 gives an overview. We conclude that "everything can be done in CHR":

Corollary 1. *For every (RAM machine) algorithm which uses at least as much time as it uses space, a CHR program exists which can be executed in the Leuven CHR system in hProlog, with time and space complexity within a constant from the original complexities.*

4 Discussion and Conclusion

One may expect to pay *some* performance penalty for using a very high-level language like CHR. Therefore, it is good to have a complexity-wise completeness result, which essentially proves that one can always get the asymptotic time and space complexity right in CHR. Of course the constant factors are also important in practice. These have also been investigated in [7]. The general construction described above (using a RAM machine simulator) predictably yields very large

constant factors — about four orders of magnitude between CHR(hProlog) and assembler code. However, using more elegant high-level CHR programs to implement an algorithm, a much more acceptable performance was measured: the CHR(hProlog) programs (for Union-find and Dijkstra's algorithm with Fibonacci heaps) were 'only' about one order of magnitude slower than direct implementations in C, and they used about 3 to 10 times as much space.

For other declarative programming languages, it remains a challenge to prove a similarly strong complexity-wise completeness result. In [7], a first attempt was made to "port" the result to some other declarative languages. For Prolog, Haskell, and Maude [11] we could not find a way to achieve complexity-wise completeness within the pure fragment of the language. In Jess [12] we did get complexity-wise completeness, but with a much worse constant factor (about 30 times slower than CHR). Other languages still have to be investigated.

Acknowledgments. This research was funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Most of the work presented in my Ph.D. thesis was joint work with Tom Schrijvers, Bart Demoen, Peter Van Weert, and Leslie De Koninck.

References

1. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009)
2. Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As time goes by: Constraint Handling Rules — A survey of CHR research between 1998 and 2007. Theory and Practice of Logic Programming (2009)
3. Schrijvers, T.: Analyses, optimizations and extensions of Constraint Handling Rules. PhD thesis, K.U.Leuven, Leuven, Belgium (June 2005)
4. Duck, G.J.: Compilation of Constraint Handling Rules. PhD thesis, University of Melbourne, Victoria, Australia (December 2005)
5. De Koninck, L.: Execution control for Constraint Handling Rules. PhD thesis, K.U.Leuven, Leuven, Belgium (November 2008)
6. Duck, G.J., Stuckey, P.J., García de la Banda, M., Holzbaur, C.: The refined operational semantics of Constraint Handling Rules. In Demoen, B., Lifschitz, V., eds.: ICLP '04. LNCS, vol. 3132, Springer (2004) 90–104
7. Sneyers, J.: Optimizing Compilation and Computational Complexity of Constraint Handling Rules. PhD thesis, K.U.Leuven, Leuven, Belgium (November 2008)
8. Demoen, B., Nguyen, P.L.: So many WAM variations, so little time. In: Proc. 1st Intl. Conf. Computational Logic. LNCS, vol. 1861, Springer (2000) 1240–1254
9. Sneyers, J., Schrijvers, T., Demoen, B.: The computational power and complexity of Constraint Handling Rules. ACM Trans. Program. Lang. Syst. **31**(2) (2009)
10. Frühwirth, T.: As time goes by: Automatic complexity analysis of simplification rules. In: Proc. 8th Intl. Conf. Princ. Knowledge Representation and Reasoning, Morgan Kaufmann (2002) 547–557
11. Clavel, M., Durán, F., Eker, S., et al.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science **285**(2) (2002) 187–243
12. Friedman-Hill, E.: Jess in Action: Java Rule-Based Systems. Manning Publications (2003)