The K.U.Leuven CHR system: implementation and application

Tom Schrijvers[1]*, Bart Demoen[1]

Dept. of Computer Science, K.U.Leuven, Belgium

Abstract. We present the K.U.Leuven CHR system: what started out as a validation of a new attributed variables implementation, has become a part of three different Prolog systems with an increasing userbase.
In this paper we highlight the particular implementation aspects of the K.U.Leuven CHR system, and a few CHR applications that we have built with our system.

1 Introduction

For this paper we expect the reader to be familiar with Constraint Handling Rules in general and their implementation on Prolog systems in particular. We refer the reader to [6, 7] and [9] for more background on the respective topics.

CHR has been around for several years now, but the number of CHR implementors and variety of available implementations is surprisingly small. The best known CHR implementation is the one within SICStus Prolog [10]. It was also ported to Yap [1]. We are familiar with only one different implementation, namely the one in ECLiPSe [8], but which seems to be lacking features, like multi-headed propagation rules.

Our work adds one more CHR Prolog implementation to the shortlist. The host system was originally hProlog, but it is now also available in XSB [16] and SWI-Prolog [17].

In the next Section, we present our CHR system: its evolution, current state and particular implementation aspects. Next, in Section 3 we mention several of our applications for which CHR has proven to be the right tool. Two of them are discussed in more detail. Finally, in Section 4 we discuss future work on our CHR compiler, challenges we see for the language and application domains we wish to tackle with CHR.

2 Implementation

The K.U.Leuven CHR system consists of two parts:

- The *runtime* is strongly based on the SICStus CHR runtime written by Christian Holzbaur.
- The *preprocessor* compiles embedded CHR rules in Prolog program files into Prolog code. The compiled form of CHR rules is similar to that described in [9].

Initially the CHR system described in this paper was written for the hProlog system. hProlog is based on dProlog [4] and intended as an alternative backend to HAL [3] next to the current Mercury backend. The initial intent of the implementation of a CHR system in hProlog was to validate the underlying implementation of dynamic attributes [2].

After the completion of the main functionality of the CHR system, it was ported to XSB. This port was rather straithforward because we first introduced into XSB the hProlog attributed variables interface. We have then integrated CHR with the tabled execution strategy of XSB [15] to facilitate applications with tabled constraints. In that context we have studied optimization via generalised answer subsumption of CHR constraints.

Jan Wielemaker - the implementor of SWI-Prolog - recently became interested in attributed variables as a basis for co-routining and constraint facilities in SWI-Prolog. He decided to adopt the attributed variables implementation of hProlog. Once these were in place, it required little work to also adopt the K.U.Leuven CHR system to SWI-Prolog.

Some noteworthy optimizations of the compiler are:

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

– Inference of functional dependencies between arguments of constraints that takes guards into account. This allows to stop search for passive constraints in multi-headed rules after a first solution has been found.
– Classification of left-hand constraint as passive for rules `C1 \ C2 <=> true | Body` where `C1` and `C2` are identical modulo the arguments that are functionally dependent on other arguments.
– Automatic detecting of constraints that are never inserted in the constraint store. One of the corollaries of this information, is that other constraints appearing in rules with *never inserted* constraints can be considered passive.
– Heuristic reordering of passive constraints in multi-headed rules to increase the sharing of variables and interleaving with cheap guards.
– Avoidance of redundant locks in guards.

2.1 Experimental evaluation

In this Section we compare the performance of our CHR systems with that of Christian Holzbaur on their respective systems. To do so we compare the results of eight benchmarks which are available from [13].

The following causes for performance differences are to be expected:

– Firstly, we expect the outcome to be mostly determined by the relative performance difference on Prolog code as the CHR rules are compiled to Prolog. For plain Prolog benchmarks, we have found average runtimes of 78.2 % for Yap, 76.2 % for hProlog, 146.8 % for XSB and 488.2 for SWI-Prolog. These times are relative to SICStus.
– Secondly, the results may be influenced by the slightly more powerful optimizations of our CHR preprocessor. To eliminate these effects we have disabled all advanced optimizations not performed by the SICStus CHR compiler. In addition, the checking of guard bindings has been disabled in both systems. This does not affect the benchmarks, since no binding or instantiation errors occur in the guards. This increases the fairness of comparison since our analysis of redundant checks is more powerful and at the same time our system does not intercept instantiation errors.
– Thirdly, the low-level implementation and representation of attributed variables differs between the systems. The global constraint store of CHR is represented as an attributed variable and it may undergo updates each time a new constraint is imposed or a constraint variable gets bound. Hence, the complexity and efficiency of accessing and updating attributed variables may easily dominate the overall performance of a CHR program if care is not taken. Especially the length of reference chains has to be kept short and nearly constant, as otherwise accessing the cost of dereferencing the global store may easily grow out of bounds.

Table 1 shows the results for the benchmarks. All measurements have been made on an Intel Pentium 4 2.00 GHz with 512 MB of RAM. Timings are relative to SICStus. The Prolog systems used are SICStus 3.11.0 and Yap with Christian Holzbaur's CHR system on the one hand and hProlog 2.4, our extension of XSB 2.6 and SWI-Prolog 5.3 with our CHR system on the other hand.

Table 1. Runtime performance of 8 CHR benchmarks in 5 different Prolog systems.

	Christian Holzbaur		K.U.Leuven		
Benchmark	SICStus	Yap	hProlog	XSB	SWI-Prolog
bool	100.0%	74.1%	43.3%	86.0%	207.0%
fib	100.0%	61.7%	76.5%	154.9%	538.9%
fibonacci	100.0%	59.6%	35.4%	82.2%	270.0%
leq	100.0%	96.0%	81.6%	151.1%	454.4%
primes	100.0%	104.8%	51.3%	139.5%	520.3%
ta	100.0%	82.6%	53.1%	106.2%	380.4%
wfs	100.0%	63.6%	52.5%	125.9%	309.3%
zebra	100.0%	52.6%	21.3%	50.8%	139.1%
average	100.0%	74.4%	51.9%	112.1%	352.4%

We see that the relative performance difference between SICStus and Yap is more or less the same for both CHR and plain Prolog. On the other hand, the relative speed for CHR compared to SICStus is 1.31 to 1.47 times better than that of plain Prolog for the other three systems. We believe that this is partly due to the more efficient implementation of attributed variables. Another important factor is the code quality: we have found that careful tuning of the generated code and runtime system, based on profiling results, considerably improves runtimes.

3 Applications

Our experience with CHR is not limited to developing the system, we have also experienced the power of the CHR language itself in several applications.

In previous work [15] we have already reported on two model checking applications where a rapid CHR prototype has replaced the adhoc constraint manipulations.

We have also assisted master thesis students in writing a simple formula simplifier in CHR as part of a precondition checker for Object Oriented programs.

In the following we will give a brief overview of two other applications, a wellfounded semantics generator and a generative memory model implementation.

3.1 Wellfounded Semantics Generator

To illustrate the power of CHR for general purpose applications, we have implemented an algorithm that computes the wellfounded semantics of simple logic programs. Simple logic programs consist of clauses with an atom in the head and both positive and negated atoms in the body.

The algorithm proceeds by repeated application of two steps until nothing changes anymore. In the first step as many atoms as possible are decided to be either true or false. In the second step, only the clauses of the still undefined atoms are considered. In those clauses only the positive body literals of atoms not previously proven true are considered. All those remaining atoms that cannot be proven true under those circumstances are considered false in the wellfounded semantics. We carry this information over to step one and continue.

This algorithm is quite a challenge for CHR:

- The sequencing of the two different steps has to be enconded.
- The second step only considers a subset of the information. It disregards the negative literals.
- The information inferred in the second step has a different meaning in the first step. Undefined atoms in the second step are considered false for the first step.

To tackle this problem, we heavily rely on the actual operational semantics of CHR: the order of the rules is important for the correct execution of the program.

Several typical CHR idioms used are:

- Use special flag constraints to indicate a certain step is active and to enable certain CHR rules.
- If-then-else through order of rules: the first rule removes key constraint if it commits. The second rule applies if key constraint is still around.
- Effect change of semantics by replacing constraints with others, e.g. `pos/2` with `pos2/2`, such that other rules apply to them.

This program, called `wfs`, is available from [13]. It shows that the impure features of CHR can be put to good use, even to compute pure logic results. The fact that CHR permits these kinds of techniques considerably improves its usability and suitability for a wider range of problems. However, more syntactical support for alternating between phases with different semantics, e.g. by enabling and disabling certain rules, would be helpful.

3.2 JmmSolve: a generative CCM Machines implementation

Because the current Java memory model shows several flaws, JSR-133 calls for a new memory model for Java. A memory model describes the allowed interactions between different threads via main memory, in particular the relation between reads and writes to main memory.

One proposal is the Concurrent Constraint-based Memory Machines [12] framework which allows the expression of several memory models in terms of constraints. The two main CCM concepts are events, like reads, writes, locks etc., and constraints on events. Constraints on events are either ordering constraints or constraints on values read or written.

In a CCM Machine threads submit events and corresponding constraints to the constraint store which functions as the main memory that merges them with the previous submissions. According to the rules of the underlying memory model the CCM then adds ordering constraints between events, links reads to writes, and checks *the unique solution criterion* of the linking, i.e. whether every value read and written has one unique instantiation.

With JmmSolve [14] we prove the claim of the CCM proposal that it is generative, i.e. JmmSolve is capable of generating all valid orderings and linkings according to the rules of the memory model.

JmmSolve currently uses CHR in two places:

- For a minimalistic integer constraint solver with constraint implication and equality. This only serves as a proof of concept. We could have used just as well another full blown integer solver. However, this observation was not clear at the beginning and CHR has allowed us to go ahead without worrying over possible interoperability problems. Now that the prototype has been established, it is fairly easy to indeed improve efficiency and use a genuine integer solver.
- For the event ordering constraint `(<<)/2` together with the ordering and linking rules of the memory model. As the ordering constraint is subject to application-specific ordering rules, this definitely calls for CHR. Indeed, it has proven to be rather easy to translate the rules of memory models into CHR rules.

We do exploit the actual operational semantics of CHR for tasks which may be considered impure in traditional constraint logic programming, such as collecting all constraint of a particular kind from the store and relying on the order in which constraints are added to the program.

JmmSolve was released and is currently under scrutiny of the Java Memory Model community.

4 Future Work

Future work on our compiler will investigate more optimizations. In particular, we are considering the usefulness of intelligent backtracking techniques, perhaps combined with backmarking. In addition, ideas of other rule-based languages, like the Rete algorithm [5], may be adapted to the CHR setting.

Because of the current wider acceptance of CHR we see as one of the more important challenges of the CHR community the standardisation of syntax, features and operational semantics. Toghether with its availability on a wide range of popular Prolog systems a single standard should make CHR a good candidate for acceptance in the Next Logic Programming Language, about which discussion have recently started [11].

Two of the issues that should be considered in a standard are:

- CHR currently seems to lack structuring features, which are essential for programming in the large. The requirement of multiple phases in Section 3.1 is an example where the need for more structuring features is revealed.
- Another issue are the options and pragmas supported by different systems. Some syntactical difference should be made between those with an impact on semantics versus efficiency. In that way, unsupported efficiency options can safely be ignored while unsupported semantical options should be reported to the user.

To support further research in optimizations for CHR, a publicly available benchmark suite with a balanced mix of interesting properties is another discussion topic for CHR implementors.

As to future CHR applications, we are interested in further investigating the usefulness of tabled CHR for model checking with constraints. We also inted to experiment with CHR for execution control based on high-level application specific constraints in multi-agent systems.

Acknowledgements

We thank David S. Warren and Jan Wielemaker for their cooperation and support of the K.U.Leuven CHR system, and Christian Holzbaur for allowing the use of his runtime system.

References

1. V. S. Costa, L. Damas, R. Reis, and R. Azevedo. *YAP User's Manual.* http://www.ncc.up.pt/~vsc/Yap/.
2. B. Demoen. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium, oct 2002. URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW350.abs.html.
3. B. Demoen, M. G. de la Banda, W. Harvey, K. Marriott, and P. J. Stuckey. An Overview of HAL. In *Principles and Practice of Constraint Programming*, pages 174–188, 1999.
4. B. Demoen and P.-L. Nguyen. So many WAM variations, so little time. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey, editors, *Computational Logic - CL2000, First International Conference, London, UK, July 2000, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000.
5. C. L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. In *Artificial Intelligence*, volume 19, pages 17–37, 1982.
6. T. Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in Lecture Notes in Computer Science, pages 90–107. Springer Verlag, March 1995.
7. T. Frühwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Special Issue on Constraint Logic Programming*, volume 37, October 1998.
8. W. Harvey et al. *ECL^iPS^e User Manual. Release 5.3.* European Computer-Industry Research Centre, Munich and Centre for Planning and Resource Control, London, 2001.
9. C. Holzbaur and T. Frühwirth. Compiling Constraint Handling Rules. In *ERCIM/COMPULOG Workshop on Constraints*, CWI, Amsterdam, 1998.
10. Intelligent Systems Laboratory. *SICStus Prolog User's Manual.* PO Box 1263, SE-164 29 Kista, Sweden, October 2003.
11. P.J.Stuckey. The straw man proposal for NLPL. In *ALP Newsletter*, volume 17, February 2004.
12. V. Saraswat. Concurrent Constraint-based Memory Machines: A framework for Java Memory Models (Preliminary Report). Technical report, IBM, March 2004.
13. T. Schrijvers. Benchmarks used for comparing CHR systems on different Prolog systems, January 2004. Available at http://www.cs.kuleuven.ac.be/~toms/Research/chr.html.
14. T. Schrijvers and B. Demoen. JmmSolve: a generative reference implementation of CCM Machines. Technical Report CW379, Katholieke Universiteit Leuven, March 2004.
15. T. Schrijvers, D. S. Warren, and B. Demoen. CHR for XSB. In R. Lopes and M. Ferreira, editors, *Proceedings of CICLOPS 2003. Technical Report DCC-2003-05, DCC - FC & LIACC, University of Porto*, pages 7–20, December 2003.
16. D. S. Warren et al. The XSB Programmer's Manual: version 2.5, vols. 1 and 2, 2001.
17. J. Wielemaker. SWI-Prolog's Home. http://www.swi-prolog.org/.