The K.U.Leuven CHR System

Tom Schrijvers
K.U. Leuven
Belgium

Editor: Roberto Bagnara

Introduction

It has been about 15 years now since the Constraint Handling Rules (CHR) [**1**] language saw the light. Since then it has become quite clear that CHR is a very powerful complement to Prolog. Both have a very firm basis in logic, but whereas Prolog is about single-headed rules, backward chaining and backtracking, CHR has multi-headed rules, forward chaining and committed choice.

CHR's features haven been chosen to support the modeling and implementation of constraint solvers. Today we know of no better way to quickly implement solvers for and gain insight in new constraint domains. In recent years, it has become clear that CHR is also useful for many other types of applications. The usual applications of (Term) Rewriting Systems and Production Rule Systems are quite easy to implement in CHR as well. Also imperative algorithms can be expressed in a high-level manner in CHR. In short, be it for logical reasoning services or for any other general purpose application, CHR is a language to consider.

CHR is usually embedded in Prolog as an add-on library and for a long time the SICStus implementation by Christian Holzbaur [**2**] has been the standard implementation. This library was also available in Yap; two different, older implementations come with ECL^{i}PS^{e}. Unfortunately, as far as we know, neither of these systems is currently still being actively developed or even maintained.

In this article we present our new implementation: the K.U.Leuven CHR system [**3**,**4**] . It provides a user-friendly and highly efficient CHR implementation to all Prolog users.

System Overview

In this section we briefly present the evolution and spreading of the K.U.Leuven CHR system and its new language features. We also take a look under the hood, at the used program analyses and optimized compilation techniques.

The K.U.Leuven CHR Diaspora

The K.U.Leuven CHR system started out as small practical project during the author's Ph.D. studies, to obtain a benchmark for Bart Demoen's new dynamic attributed variables implementation in hProlog. The output of SICStus' CHR was reverse engineered to obtain a base system. However, it became clear very soon that there was much potential for improving the system, and gradually the system diverged from its roots. It became the central topic of my Ph.D. thesis on

analyses, optimizations and language extensions of CHR [**5**]. Today the K.U.Leuven CHR team consists of 5 people actively working on the system and related research topics: Jon Sneyers, Leslie De Koninck, Peter Van Weert, Bart Demoen and myself.

In July 2004 the system was ported to XSB and integrated with tabled execution [**6**]. Half a year later Jan Wielemaker was looking for an easy way to provide general constraint solving capabilities to SWI-Prolog. The smoothness of the port to XSB convinced him that the K.U.Leuven CHR system was what SWI-Prolog needed [**7**]. From then on SWI-Prolog has been the K.U.Leuven CHR system's main supported platform, and Jan Wielemaker has taken care of tight integration with the rest of the system, notably with the debugger. Gradually other open source Prolog systems became interested in this new CHR system and its evidence of portability, and today you can find K.U.Leuven CHR in hProlog, XSB, SWI-Prolog, Yap, Ciao and B-Prolog. The system will take its first steps in a commercial system in the upcoming SICStus 4 release, where it replaces the old implementation by Christian Holzbaur.

Language Features

The K.U.Leuven system provides you with the familiar CHR syntax for writing rules embedded in a Prolog program. In addition to the usual syntax for rules, K.U.Leuven CHR offers a number of additional mandatory and optional declarations. These have been chosen for improved developer experience and to control optimization, debugging and runtime checks. Let's have a look at these declarations in the context of the well-known Fibonacci numbers computation.

```
   :- use_module(library(chr)).
 1.
 2. :- chr_option(debug,off).
 3. :- chr_option(optimization,full).
 4.
 5. :- chr_constraint fib(+int,?int).
 6.
 7.  basecase @ fib(N,F) <=> N =< 1 | F = 1.
 8. induction @ fib(N,F) <=> N > 1  | N1 is N -1, fib(N1,F1), N2 is
 9. N - 2, fib(N2,F2), F is F1 + F2.
10. redundant @ fib(N,F) <=> F = 0.
11.
12. main(N) :- fib(N,F), format('The ~wth fibonacci number is
    ~w.',[N,F]).
```

In the first line, we import the CHR library. This allows the definition of new CHR constraints and rules, but you can write ordinary Prolog code as well (e.g., line 12). But there's more to the import declaration than just making the CHR syntax available. When the file is consulted, the declaration invokes the K.U.Leuven CHR compiler behind the scenes to transform the CHR constraints into executable Prolog predicates.

The `chr_constraint` declaration in line 6 declares all the CHR constraints, in our case just the `fib/2` constraints. The declaration is mandatory for two reasons. Firstly, it is possible in this way to declare constraints that are not defined by any rules. Secondly, and more importantly, it allows the K.U.Leuven CHR system to check whether the arities in the heads of rules are consistent. For example, if we had inadvertently written `fib(N)` in line 8, the system would tell us that there is an `undeclared constraint fib/1 used in head of rule` basecase. We know from experience that this is a quite frequent bug, and that

this declaration saves a lot of time. This example also illustrates the use of having rule names: error messages are so much more meaningful when they can refer to them.

Optionally, the `chr_constraint` code allows you to write the modes and the types of the constraints you are using. This provides good documentation as well as being useful for verification (finding bugs) and optimized compilation. Taking into account the declared types and modes, the K.U.Leuven CHR figures out that the first two rules in the example are exhaustive, and that the third rule in line 10 is hence dead code. This may indicate some unintended behavior, so the user is warned that the rule will never fire. K.U.Leuven CHR offers a number of built-in types (e.g., int and float) but also allows the user to define Hindley/Milner style (possibly polymorphic) algebraic data types and type aliases, with similar syntax as in the Mercury language. For example, the following two lines respectively define a polymorphic list type and an alias for integer lists:

```
:- chr_type list(T) ---> [] ; [T | list(T)].
:- chr_type intlist == list(int).
```

Finally, additional options can be declared through the `chr_option/2` declarations. The two main options are the debugging facility (enabled by default) and optimized compilation (disabled by default). We'll say a bit more about the latter in the next section. The debugging facility is integrated with that of the underlying Prolog mechanism. Whenever you enable the Prolog tracer, you get to see the CHR execution ports (addition and removal of constraints and rule application) interleaved with those of Prolog. What is fortunately hidden, however, are the ports of the compiler generated Prolog code!

Optimized Compilation and Program Analysis

Although there have been various CHR implementations over the years, some only at the level of prototypes and others more advanced, it is safe to say that the K.U.Leuven CHR system is the most optimized one. The generated code and data structures are highly specialized for the compiled program. Examples of the latter are program-specific indexing structures based on hash-tables, linked lists and global variables.

These optimizations allow us to write imperative algorithms, such as union-find [**8**] and Dijkstra's shortest path with Fibonacci heaps [**9**], in a very concise way and yet obtain optimal time complexity. And the constant factors are not so bad either: consider for example that Dijkstra's shortest path runs only 10 times slower in K.U.Leuven CHR on hProlog than a hand-coded implementation in C. Various upcoming improvements will reduce this factor even further.

In recent work we also address the issue of space usage [**10**]. With the help of inlining, specialized constraint suspensions, static and dynamic forms of memory reuse we are able to considerably reduce the amount of memory consumption. For example, a Random Access Memory machine simulator written in K.U.Leuven CHR runs without any unnecessary memory overhead at all. Together with our results on runtime complexity we can say that any algorithm can be written with optimal space and time complexity in K.U.Leuven CHR [**11**].

A crucial aspect of the compilation process in K.U.Leuven CHR that drives the optimized compilation, are the various program analyses. K.U.Leuven CHR uses various techniques, ranging from simple pattern matching over sophisticated

theorem proving [**12**] to full scale abstract interpretation [**13**]. Of course many parts of the compiler are written in CHR itself and CHR's implicit fixpoint semantics are particularly handy for mutual improvement of analysis results. And of course, where the analysis fails the programmer can lend a hand with declarations for types, modes and passive occurrences (dead generated code).

K.U.Leuven CHR in Java

Prolog is not the only language in which CHR can be embedded: Peter Van Weert has ported K.U.Leuven CHR to the Java language. The system is named K.U.Leuven **J**CHR [**14**], and integrates the usual CHR syntax with typical OOP features that are familiar for Java programmers.

Applications and Users

It is very gratifying for a system when people find it useful for learning new things and coming up with new ideas and practical applications. As K.U.Leuven CHR is not packaged separately, but comes with various Prolog implementations, it is very difficult to know the extent of its user base and the type of applications that are built with it. Here we present a number of applications we are aware of.

Industrial Applications

SWI-Prolog is quite an interesting system for industrial applications as it has quite a number of libraries that make it easy to interface with the real world (semantic web, XML, GUI, ...). Some of its industrial users have discovered the advantages of CLP, and CHR in particular. A notable user of CHR in this context is Scientific Systems and Software LTD. This New Zealand-based company employs CHR as part of their SecuritEase stock brokering system [**15**], for handling forms and for transforming their Constraint Query Language into SQL. Other companies use K.U.Leuven CHR for production planning and even for handling rounding errors.

The INCLP($\mathbb{R}$) System

The biggest research application of the K.U.Leuven CHR system that we are aware of is the INCLP($\mathbb{R}$) [**16**] by Leslie De Koninck. It is an interval-based non-linear constraint solver over the reals, in the spirit of the well-known CLR($\mathbb{R}$) system. Its interval-based nature avoids rounding errors and its non-linear constraints are essential for dealing with the real world: non-linearity appears in many sciences (e.g., biology), economics,... CHR allows the system to quickly experiment with different existing algorithms for, e.g., consistency and to explore new variants.

Type Inference

It has already been shown by several people that CHR is an excellent language for type inference and type checking. You can easily write your type domain as a logical theory and directly map its axioms onto CHR rules. This highly declarative approach trivially guarantees soundness. Variations of the type system come down to changing a few axioms/rules. We have applied this approach to the work of Bruynooghe and Gallagher on type reconstruction for Prolog: polymorphic instances are now supported, polymorphic recursion is dealt with based on a formalization of Fritz Henglein's algorithm as a constraint theory, and various extensions are explored [**17**]. In current research we are

considering a general formalization of type inference for Generalized Algebraic Data Types.

JmmSolve

Several flaws had been discovered in Java's previous (thread-) memory (interaction) model. A new memory model was called for, and one of the proposals was Vijay Saraswat's framework of Concurrent Constraint-based Memory Machines (CCMMs). His idea was to have a declarative model based on the concurrent constraint programming paradigm, which would be more intuitive for programmers and at the same time simplify formal reasoning. We have used CHR in this context as a prototyping language for implementing an executable CCMM [**18**]. With only small changes to the rules, different CCMM instances can be explored within the framework. Unfortunately, the Java committee decided to go with a more traditional, imperative memory model.

Conclusion

We have introduced to you the K.U.Leuven CHR system, a user-friendly and highly efficient CHR system for Prolog. It is available in all major open source Prolog systems and actively being maintained by the K.U.Leuven CHR team. Many applications in industry, research and education are using the system. The implementation of the system itself also serves as a basis for research in program analysis and optimized compilation for rule-based languages.

It is our ongoing effort to improve the scalability and user-friendliness of the system. Although there have been major improvements in the last few years, much lies still ahead of us. So we do encourage the Logic Programming community, from implementors to researchers, to join us in our effort. Give us your feedback, or why not even dig in yourself to extend the system or test your ideas. We are happy to help you.

Have you become curious about the K.U.Leuven CHR system and would you like to experiment with it? Chances are that you already have it! Just look at the documentation of your favorite Prolog system. If it's not there, do ask the developers why it's missing (or consider one of the excellent open source Prolog systems).

Acknowledgments

I would like to thank all the people that have directly or indirectly contributed to the K.U.Leuven CHR system: Christian Holzbaur, David S. Warren, David Trallero, Gregory J. Duck, Jan Wielemaker, Manuel Hermenegildo, Mike Elston, Neng-Fa Zhou, Peter J. Stuckey, Thom Frühwirth, Vitor Santos-Costa and many others.

Bibliography

1. T. Schrijvers and T. Frühwirth, **The CHR website**.
2. C. Holzbaur and T. Frühwirth, A Prolog Constraint Handling Rules Compiler and Runtime System, Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules (C. Holzbaur and T. Frühwirth, Eds.), Taylor & Francis, Vol 14(4), April 2000.
3. T. Schrijvers, and B. Demoen, The K.U.Leuven CHR system: Implementation and application, First Workshop on Constraint Handling Rules: selected contributions (Frühwirth, T. and Meister, M., eds.), pp. 1-5, 2004.
4. T. Schrijvers, **The K.U.Leuven CHR system website**.

5. T. Schrijvers, Analyses, optimizations and extensions of Constraint Handling Rules, Ph.D. Thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, June, 2005, xxiv+210 pages.
6. T. Schrijvers, and D. Warren, Constraint Handling Rules and Tabled Execution, Logic Programming, 20th International Conference, ICLP 2004, Proceedings (Demoen, B. and Lifschitz, V., eds.), vol 3132, Lecture Notes in Computer Science, pp. 120-136, 2004.
7. T. Schrijvers, J. Wielemaker and B. Demoen, Constraint Handling Rules for SWI-Prolog, 9th Workshop on (Constraint) Logic Programming, Abstract, Ulm, February, 2005.
8. T. Schrijvers, and T. Frühwirth, Optimal Union-Find in Constraint Handling Rules, Theory and Practice of Logic Programming 6 (1&2), pp. 213-224, 2006.
9. J. Sneyers, T. Schrijvers, and B. Demoen, Dijkstra's algorithm with Fibonacci heaps: an executable description in CHR, Proceedings of the 20th Workshop on Logic Programming (Fink, M. and Tompits, H. and Woltran, S., eds.), pp. 182-191, 2006.
10. J. Sneyers, T. Schrijvers, and B. Demoen, Memory reuse for CHR, Proceedings of Twenty Second International Conference on Logic Programming, Seattle, Washington, USA (Etalle, S. and Truszczynski, eds.), 2006, accepted.
11. J. Sneyers, T. Schrijvers, and B. Demoen, The Computational Power and Complexity of Constraint Handling Rules, Proceedings of the 2nd Workshop on Constraint Handling Rules (Schrijvers, T. and Frühwirth, T., eds.), pp. 3-17, 2005.
12. J. Sneyers, T. Schrijvers, and B. Demoen, Guard and continuation optimization for occurrence representations of CHR, Logic Programming, ICLP 2005, Proceedings (Gabbrielli, M. and Gupta, G., eds.), vol 3668, Lecture Notes in Computer Science, pp. 83-97, 2005.
13. T. Schrijvers, P. Stuckey, and G. Duck, Abstract interpretation for constraint handling rules, Proceedings of the Seventh ACM SIGPLAN Symposium on the Principles and Practice of Declarative Programming (Felty, A., ed.), pp. 218-229, 2005.
14. P. Van Weert, **The K.U.Leuven JCHR System website**.
15. Scientific Software & Systems LTD., **The SecuritEase website**.
16. L. De Koninck, **The INCLP(ℝ) System website**.
17. T. Schrijvers, and M. Bruynooghe, Polymorphic algebraic data type reconstruction, Proceedings of Eigth ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming, Venice, Italy (Maher, M., ed.), 2006, accepted.
18. T. Schrijvers, JmmSolve: A generative Java Memory Model implemented in Prolog and CHR, Poster Presenation, Proceedings of The 20th International Conference on Logic Programming, St.-Malo, France (Demoen, B. and Lifschitz, V., eds.), September 2004.