A Literate Programming System for Logic Programs with Constraints

Eduardo Torres Schumann

Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilian University Munich, Oettingenstr. 67, 80538 Munich, Germany
torres@cis.uni-muenchen.de

Abstract. In this article we present a prototype of a literate programming system for CLP programs. We choose LaTeX as documentation language and, by introducing special annotation tags, generate constraint logic programs from documents that specify them. We propose the use of Constraint Handling Rules (CHR) for the generated programs, as this language has shown great generality and flexibility when specifying and implementing constraint solvers. With our annotation method, we do not restrict the set of LaTeX commands that can be used in the program specification, but give the system some information about them in order to generate runnable programs. This provides mainly two advantages: documents that were not explicitely written for this system can be easily adapted to generate CHR implementations, and a special syntax for program specifications can be defined and used subsequentely in several documents, saving some annotation work this way. We also give an example of program specification in LaTeX that originally motivated the development of the system and annotate them with our special tags. A CHR implentation for it can be extracted with our system from the present article.

1 Literate Programming and Constraint Logic Programming

In [1, 2], Knuth states his idea of *Literate programming*, which attends to solve the software documentation problem and offers a methodological basis to document programs. While traditionally computer programs are documented with scattered comments in the code and presented in the machine execution order, in a *literate* program, the emphasis is reversed. The programmer writes a text mainly directed at humans in which the construction of the program is described in the order of the design process, making this description precise and telling the computer what to do with pieces of actual code inserted in the text. Therefore a typesetting language and a programming language are integrated in a literate programming system, so that a flexible combination of program and documentation parts in a single document is possible. Usually, tools provided by a literate programming system extract the code and generate the documentation from the input file and also may offer some facilities for debugging or performing verifications.

On the other hand, programs written with Constraint Logic Programming (CLP) languages are relatively short pieces of code that achieve a high level of expressiveness. Since they are written in a highly declarative language, they are closer to a specification than a traditional program. The aspiration of the *Literate Programming* paradigm of developing a machine understandable program starting out from a clear exposition of ideas appears here naturally attainable.

There is a CLP language that takes advantage of using a typesetting language to express programs. NCL (Natural Constraint Language) [14] offers a unified framework for solving Boolean, integer and set constraints. It completely adopts a mathematical notation expressed in TEX mathematical mode, thus making possible to write the code and the documentation without a transition. But NCL is not an universal constraint solver.

The programming language *CHR (Constraint Handling Rules)* [3] was originally introduced to ease the development of constraint solvers. Any first-order constraint theory or consistency algorithm can be implemented at a high level of abstraction. The usual formalisms to describe a constraint theory, i.e. inference rules, rewrite rules, sequents, first-order axioms, can be expressed as CHR rules in a straightforward way. Starting from this executable specification, the rules then can be refined and adapted to the specifics of the application. Over time it has become apparent that CHR and its extensions are useful for implementing other classes of algorithms, especially in computational logic: combining deduction, abduction and constraints, combining forward and backward chaining, bottom-up evaluation with integrity constraints, top-down evaluation with tabulation, and in general, production rule systems. This broad application spectrum makes CHR an ideal candidate to form part in a CLP literate programming system in addition to a documentation language.

In our approach, we choose a middle course between alternating description and code pieces in *Literate Programming* style and completely integrating the programming syntax with the documentation language. We generate an implementation, a CHR program, from a formal specification typeset in LATEX. The formal specification is not bounded to a specific syntax, so some supplementary annotations are used in order to ensure that the extracted program is executable.

One advantage of this procedure is that programs can also be generated from documents that were not written with this system in mind by just providing some annotations. In fact, the idea for the system appeared when trying to implement with CHR classical algorithms that were specified in a declarative, logical form in order to analyze its properties. An original example is presented in Sect. 4. Another advantage is that one can also define his own special LATEX specification language for CHR programs, wich can then be presented in a mathematical notation. In our prototype, annotated documents can refer to such a definition, reducing the amount of annotations needed.

In the following sections, we give an overview on CHR, describe our system and present some examples that motivated the development of it.

2 An Overview of CHR

2.1 An Introductory Example

A *constraint* is an atomic formula in first-order logic. We distinguish between *built-in (predefined) constraints* and *CHR (user-defined) constraints*. Built-in constraints are those handled by a predefined, given constraint solver. CHR constraints are those defined by a CHR program.

A CHR program consists of simplification and propagation rules. Simplification replaces constraints by simpler constraints while preserving logical equivalence. Propagation adds new constraints which are logically redundant but may cause further simplification.

In the following CHR program example, we define a order relation `X =< Y` as a constraint, while = is a built-in equality constraint:

```
X =< Y <=> X = Y | true.     % reflexivity
X =< Y,Y =< X <=> X = Y.     % antisymmetry
X =< Y,Y =< Z ==> X =< Z.   % transitivity
```

The reflexivity rule states that `X =< Y` is true if the guard `X = Y` holds. Each occurrence of `A =< A` is replaced by `true`. The antisymmetry rule simplify the constraints `X =< Y` and `Y =< X` to `X = Y`. This rule has no guard and can be applied always. Both rules are *simplification* rules, because they simplify or solve constraints. The transitivity rule states that the conjuction of `X =< Y` and `Y =< Z` implies `X =< Z` and propagates this constraint, that can be used as logical consequence to solve other constraints. E.g, when dealing with the constraints `A =< B,C =< A,B =< C`, the transitivity rule can be applied on `C =< A,A =< B`. The constraint `C =< B` is propagated and then simplified within the antisymmetry rule together with `B =< C` to `B = C`. Since now `B` equals `C`, the antisymmetry rule can be applied another time replacing `A =< B,C =< A` by `A = B`. As result we obtain `A = B` and `B = C`.

Apart of simplification and propagation rules, there is another hybrid type of rules in CHR, *simpagation* rules. We introduce them in the next section, where we present the syntax and semantics of CHR.

2.2 Syntax and Semantics of CHR

Syntax In the following, upper case letters stand for conjunctions of constraints.

A *CHR program* is a finite set of CHR. There are two kinds of CHR. A *simplification CHR* is of the form

```
[N '@'] H '<=>' [G '|'] B.
```

and a *propagation CHR* is of the form

```
[N '@'] H '==>' [G '|'] B.
```

where the rule has an optional name N followed by the symbol @. the left hand side (lhs) H is a conjunction of CHR constraints. The guard G followed by the symbol $|$ is a conjunction of built-in constraints. A trivial guard of the form *true* $|$ may be dropped. `true` is a built-in constraint that is always satisfied. The right hand side (rhs) of the rule consists of a conjunction of CHR and built-in constraints B.

A *simpagation CHR* is a combination of the above two kinds of rule, it is of the form

```
[N '@'] H1 '\' H2 '==>' [G '|'] B.
```

where the symbol `'\'` separates the lhs constraints into two nonempty conjunctions *H1* and *H2*.

In this paper, a simpagation rule can be regarded as efficient abbreviation of the corresponding simplification rule:

```
[N '@'] H1, H2 '<=>' [G '|'] H1, B.
```

Operational Semantics The operational semantics of CHR programs is given by a state transition system. The semantics uses interleaving for the parallel construct of conjunction. With *derivation steps (transitions, reductions)* one can proceed from one state to the next. A *derivation* is a sequence of derivation steps.

A *state (or goal)* is a conjunction of built-in and CHR constraints. An *initial state (or query)* is an arbitrary state. In a *final state (or: answer)* either the built-in constraints are inconsistent or no new derivation step is possible anymore.

Let P be a CHR program for the CHR constraints and CT be a constraint theory for the built-in constraints. The transition relation $\longmapsto$ for CHR is as follows:

Simplify
$H' \wedge D \longmapsto (H = H') \wedge G \wedge B \wedge D$
if $(H \Leftrightarrow G \mid B)$ in P and $CT \models \forall (D \rightarrow \exists \bar{x}(H = H' \wedge G))$

Propagate
$H' \wedge D \longmapsto (H = H') \wedge G \wedge E \wedge B \wedge H' \wedge D$
if $(H \Rightarrow G \mid B)$ in P and $CT \models \forall (D \rightarrow \exists \bar{x}(H = H' \wedge G))$

When we use a rule from the program, we will rename its variables using new symbols, and these variables form the sequence $\bar{x}$. A rule with lhs H and guard G is *applicable* to CHR constraints H' in the context of constraints D, when the condition holds that $CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$. Any of the applicable rules can be applied, but it is a committed choice, it cannot be undone.

If a simplification rule $(H \Leftrightarrow G \mid B)$ is applied to the CHR constraints H', the **Simplify** transition removes H' from the state, adds the rhs B to the state and also adds the equation $H = H'$ and the guard G. If a propagation rule $(H \Rightarrow G \mid B)$ is applied to H', the **Propagate** transition adds B, $H = H'$

and G, but does not remove H'. Trivial non-termination is avoided by applying a propagation rule at most once to the same constraints [13].

We now discuss in more detail the rule applicability condition $CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$. The equation $(H = H')$ is a notational shorthand for equating the arguments of the CHR constraints that occur in H and H'. More precisely, by $(H_1 \wedge \ldots \wedge H_n) = (H_1' \wedge \ldots \wedge H_n')$ we mean $(H_1 = H_1') \wedge \ldots \wedge (H_n = H_n')$, where conjuncts can be permuted.

By equating two constraints, $c(t_1, \ldots, t_n) = c(s_1, \ldots, s_n)$, we mean $(t_1 = s_1) \wedge \ldots \wedge (t_n = s_n)$. The symbol $=$ is to be understood as built-in constraint for syntactic equality (defined in CT).

Operationally, the rule applicability condition can be checked as follows: Given the built-in constraints of D, try to solve the built-in constraints $(H = H' \wedge G)$ without further constraining (touching) any variable in H' and D. This means that we first check that H' matches H and then check the guard G under this matching.

In most CHR implementations, the execution of any CHR program will result in an on-line (incremental) algorithm and any-time (approximation) algorithm. The first property means that constraints can be added during run-time, the second property means that execution can be stopped and resumed after any rule application, and the intermediate results obtained will approximate the final results. A more detailed description of CHR can be found in [4]. .

3 The Literate Programming System

Our CHR Literate Programming System consists mainly of two programs, a Prolog program, `literate.pl`, for the interaction with the user, and a Perl program, which extracts the CHR programs form the LaTeX source. We used Sicstus Prolog, because it provides a widely used CHR library, and Perl for its capabilities for processing text files, particularly its integrated regular expressions library.

`literate.pl` defines predicates that allow the user to start the extraction of the CHR programs in the source and to generate and display the corresponding Post Script file, all this from the Prolog shell. In this way, using the programs and reading the documentation can be done simultaneously.

`extract.px` generates the CHR programs from regions that were marked with special tags in the document. Another set of tags is used to perform replacements, in order to transform each defined region into a code piece compilable under Sictus Prolog. We first give an annotation example an then define the tags.

3.1 A Literate CHR Program

We could have stated our first CHR program as follows:

$\mathrm{X} \leq \mathrm{Y} \wedge \mathrm{X} \doteq \mathrm{Y} \leftrightarrow \mathrm{X} \doteq \mathrm{Y}$

$\mathrm{X} \leq \mathrm{Y} \wedge \mathrm{Y} \leq \mathrm{X} \leftrightarrow \mathrm{X} \doteq \mathrm{Y}$

$\mathrm{X} \leq \mathrm{Y} \wedge \mathrm{Y} \leq \mathrm{Z} \rightarrow \mathrm{X} \leq \mathrm{Z}$

This can be regarded as a specification for a CHR close to logic that does not consider some implementation aspects as for example, which constraints are built-in and which are user defined. Observe that there are no guards in the rules. We have annotated the document in the following way:

```
%&chr-begin
%&name: leq
%
%constraints leq/2.
%operator( 700, xfx, leq).
%&trans: leq  \leq
%&trans:  <=>  \leftrightarrow
%&trans:   ==>  \rightarrow
%&trans:   =  \doteq
%&trans:  , \land
%&built-in: =
\begin{tabular}{l}
{}X \(\leq\) Y \(\land\) X \(\doteq\) Y
                          \(\leftrightarrow\)  X \(\doteq\) Y  \\
{}X \(\leq\) Y \(\land\) Y \(\leq\) X
                         \(\leftrightarrow\)  X \(\doteq\) Y\\
{}X \(\leq\) Y \(\land\) Y \(\leq\) Z
                          \(\rightarrow\)  X \(\leq\) Z
\end{tabular}
%&chr-end
```

All tags begin with `&` and are set after a `%`, since in LaTeX, all text in a line after this character is a comment. We give a more detailed description of the tags in the next section, but, as one can think, `&chr-begin` and `&chr-end` mark the beginning and end of a region. `&trans:` tags are used to perform necessary replacements to adapt the used syntax to the CHR in Sicstus Prolog and `&built-in:` states which constraints have to be moved into the guard of the rule. `{}` mark the beginnings of the rules.

Subsequently, we let `extract.px` generate a program that implements a constraint solver for $\leq$, here represented by `leq`. This program is runnable with Sictus Prolog.

```
:- use_module( library(chr)).

handler leq.

constraints leq/2.
operator( 700, xfx, leq).

X leq Y , X = Y <=> X = Y | X = Y .
X leq Y , Y leq X <=> X = Y.
X leq Y , Y leq Z ==> X leq Z.
```

The first two lines are added by `extract.px` and forms part of a special format for `chr` regions.

3.2 Annotating and Transforming LaTeX Sources

The `extract.px` program extracts code pieces from the marked regions in the text. There are two types of regions, `chr` and `sample` regions.

Some heuristics are used to handle different LaTeX environments like `verbatim`, `alltt` and `tabular` and mathematical mode marking brackets.

In addition to the transformations triggered by the different environments, replacements can be performed in the region in order to generate a CHR program. This is annotated with the tags `&default:` and `&trans:`, see below.

Regions Regions are marked as follows

```
%&<region-type>-begin
%&name:  <name>
%
      .....

%&<region-type>-end
```

Here is `<region-type>` one of `chr` and `sample`. Text sections marked as `chr` regions are supposed to enclose statements that can be translated into Constraint Handling Rules. `sample` regions are reserved for text statements that can be interpreted as an example query. A query is here a set of constraints to be handled by the constraint solver. Each sort of region is stored after transformation in a special format, see the table below.

The `&name:` tag is used to give a name for the file generated from the region. The name for `chr` regions is also the name for the corresponding CHR handler. For `sample` regions the name can have the form of a functor acting over some variables, see Table 1. This functor can be used to call the sample constraint set as a query in the the Prolog shell.

After the `&name:` tag, other tags or LaTeX comments can follow. In the last case, the comments are also written to the CHR file if they contain Prolog declarations.

Tags In addition to the tags discussed in the regions section, the following tags are defined:

- The `&default:` and `&trans:` tags are used to define translation pairs. A statement of the form

  ```
  % &default: <translation>  <token>
  ```

Region Type	`&name:`	File Name	Output File Format
`chr`	*<name>*	*<name>*`.pl`	`:- use_module(library(chr)).` *<passed declarations>* `handler` *<name>*. *<transformed rules>*
`sample`	*<functor(A,B,..)>*	*<functor>*`.pl`	*<passed declarations>* *<functor(A,B,...)>*`:-` *<transformed constraints>*

Table 1. Region formats

will replace every `token` occurrence inside each region by `translation`. `token` is intended to be a counterpart of a Prolog atom or term, or a constraint operator. `&default:` tags can be defined in a separate file and loaded with a special predicate, so that they hold for all documents. This mechanism allows to define different syntaxes that hold for different document types. `&trans:` provides the same functionality but they have as scope the region in which there were defined.

- The `&rule-head:` can be used in a region to define a pattern that matches rule names or other strings indicating the beginning of a new rule. It can be useful to mark the starting of rules, in environments other than `verbatim` and `alltt`, in which the line structure of the region is not maintained during the transformation.
- With the `&built-in:` tag, a list of functors or constraint terms can be specified, that are then considered to be built-in constraints. They are moved into the guards of rules when performing the transformation.

```
%&built-in:  term1 , term2, ...
```

- Lines beginning with `&comment:` are passed to the Prolog program and printed to the standard output.

At the present, our transforming algorithm has a special notation for simpagation rules built in, but defining a tag to customize it is conceivable and may be part of future work. If the rule has terms between $\langle$ and $\rangle$, it is rewritten with those terms right to the `'\'`. E.g., the rule $\langle X\ leq\ Y\rangle, Y\ leq\ Z \Leftrightarrow true$ is transformed into `Y leq Z \ X leq Y <=> true.`

4 Example: The Cocke-Younger-Kasami (CYK) Algorithm

The algorithm that we are going to examine in this section is often presented in a procedural form [10], but in [11] it is specified in a bottom-up, logic form. We have implemented it by just annotating this specification. The resulting program is shown below.

The Cocke-Younger-Kasami (CYK) algorithm is a parsing algorithm that works with grammars that are in the Chomsky normal form. These are grammars with rules of the form $A \longrightarrow BC$ or $A \longrightarrow a$, where A, B, C are non-terminal symbols and a is from the terminal alphabet, but not the null character ϵ.

The algorithm parses strings bottom-up. Like other dynamic programming algorithms it stores partial results in a table.

We specify the algorithm with following rules:

(term) $$\frac{U \longrightarrow A, \quad input([A|R])}{parses(U,[A|R],R).}$$

(substr) $$\frac{input([A|R])}{input(R).}$$

(non-term) $$\frac{A \longrightarrow B \circ C, \quad parses(B,I,J), \quad parses(C,J,K)}{parses(A,I,K)}$$

The input to the parsing algorithm is a relation of grammar rules and a string to parse. A rule is represented by a constraint of the form $A \longrightarrow \ldots$, the input string is a list of characters stated within *input* constraints. Rule (*substr*) split them in suffixes that are also stored as *input* terms. Rule (*term*) tries then to assign each character a non terminal category and stores results as *parses* constraints.

A constraint $parses(A, P, Q)$ means the list difference of P and Q is of category A, where Q is a suffix of P and P an input string or one of its suffixes. In *parses* constraints that are a product of applying (*term*), the difference of P and Q is a single character. Since the input string and all its suffixes are stored as constraints, every input string position is considered by the (*term*) rule. A character may belong to several categories.

(*non-term*) takes two parsed substrings and a rule with non terminal symbols to assign a category to the concatenation of the two substrings. This rule can be applied as many times as necessary until all parse alternatives for the substrings and the input string are found. One can see the algorithm as constructing a table that stores all parse alternatives for each substring and progressively merges these results.

Input string and all suffixes are kept till the end, because they may get several categories assigned at different computation times. Grammar rules have also to be reachable for the whole time of computation. Parse results are needed for subsequent analysis and as final results. Most commonly, one is interested in knowing all possible parses. Several strings can be simultaneously parsed. This is particularly efficient if they have common suffixes, since they share *input* terms.

After properly annotating the specification above, we can extract the following program with our system:

```
:- use_module( library(chr)).

handler cyk.

operator( 900, xfx, '--->' ).
operator( 800, xfx, '°').
constraints (--->)/2, input/1, parses/3, (°)/2.

term @ U ---> A, input([A|R]) ==> parses(U, [A|R], R ).
substr @ input([A|R]) ==> input( R ).
non-term @ A ---> B ° C, parses( B, I, J), parses( C, J, K)
           ==> parses(A, I, K).
```

5 Conclusions

In this article we have presented a prototype for a Literate Programming System based on CHR that makes it possible to generate programs from mathematical notations or pseudo code than can be expressed in LaTeX with a great syntactic variability. It allows to document a broad number of applications due to the flexibility of CHR.

The relatively simple replacement mechanism that generates programs from sources is intuitive, but clearly has limitations for programs that are specified using disjunctions, existential quantifiers, since these are not directly implementable in CHR. Still, our system can be used to produce well-formed CHR programs that subsequently can undergo a source-to-source transformation. We have tested our system with examples from description logics, linear systems solvers and many others.

References

1. Knuth, Donald, *Literate Programming*, CSLI lecture notes, Stanford, CA, Center for the study of language and information, 1992.
2. Knuth, Donald, *Literate Programming* , The Computer Journal, Vol. 27, No. 2, 1984.
3. T. Frühwirth, *Theory and Practice of Constraint Handling Rules*, Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriot, Eds.), Journal of Logic Programming, Vol 37(1-3), pp 95-138, October 1998.

4. T. Frühwirth and S. Abdennadher, *Constraint-Programmierung*, Textbook, Springer Verlag, 1997.
5. T. Frühwirth and S. Abdennadher, *Essentials of Constraint Programming*, Textbook, Springer Verlag, to appear 2002.
6. C. Holzbaur and T. Frühwirth, *Constraint Handling Rules Reference Manual for Sicstus Prolog*, Technical Report TR-98-01, Österreichisches Forschungsinstitut für Artificial Intelligence, Vienna, Austria, July 1996
7. P. Deransart, R. Da Silva Bigonha, P. Parot, M. A. Da Silva Bigonha, J. De Siqueira, *A Literate Logic Programming System*, in: I Simpósio Brasileiro de Linguagens de Programação, SBLP'96, R. S. Bigonha (réd.), ANAIS, Belo Horizonte, MG, Brazil, September 1996.
8. P. Deransart, R. Da Silva Bigonha, P. Parot, M. A. Da Silva Bigonha, J. De Siqueira, *A Hypertext Based Environment to Write Literate Programs*, Proceedings of the Poster Session at JICSLP'96, N. Fuchs, U. Geske (réd.), 296, GMD-Forschungszentrum, Informationstechnik GMBH, Berlin, Septembre 1996.
9. Kopka, Helmut, LaTeX- Eine Einführung. Addison Wesley, Bonn, 1994.
10. Hopcroft, John E., Ullman, Jeffrey D., *Introduction to automata theory, languages and computation*, Addison-Wesley, 1979.
11. David A. McAllester, *On the Complexity Analysis of Static Analyses* Proceedings of the Static Analysis Symposium, 312-329, 1999.
12. H. Ganzinger, D. McAllester, *A new meta-complexity theorem for bottom-up logic programs* In Proceedings of the International Joint Conference on Automated Reasoning, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2001.
13. S. Abdennadher, Operational Semantics and Confluence of Constraint Propagation Rules, 3rd Intl Conf on Principles and Practice of Constraint Programming (CP'97), Springer LNCS 1330, 1997.
14. J. Zhou, *The Manual of NCL 1.2 (demo version)*. `http://www.loria.fr/~zhou/ziran.html`.