CHR Grammars with multiple constraint stores

Henning Christiansen

Roskilde University, Computer Science Dept.
P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: `henning@ruc.dk`

1 Introduction

CHR Grammars (CHRG) are a recent constraint-based grammar formalism added on top of CHR analogously to the way Definite Clause Grammars are defined and implemented over Prolog. A CHRG executes as an error robust bottom-up parser, and the formalism provides several advantages for natural language analysis reported elsewhere [4–6].

A notable property is CHRG's inherent ability to handle abduction in a way that requires no meta-level interpreter which otherwise is a common way to implement abduction. The technique, first noticed in [3], consists of a straightforward reformatting of grammar rules so that abducibles are passed through the constraint store, and this allows other CHR rules to serve as integrity constraints. The principle works best for grammars without local ambiguity but this is of course too limited as our main target is natural language analysis. To do abduction with an ambiguous grammar, additional techniques are necessary to avoid mixing up different sets of abducibles that represent different abductive interpretations. The last thing does not fit properly with the current paradigm of CHR, and our current implementation, which is based on impure facilities of the SICStus Prolog version of CHR [12], is not very efficient.

In the present paper we propose an architecture aimed at efficient execution of abductive CHR Grammars. It is based on a multiple constraint store model which also generalizes to committed choice languages with disjunction such as CHR$^{\vee}$ [2]; it is indicated how constraint solving in this model can be optimized using methods developed for data integration.

2 CHR Grammars and their implementation in CHR

CHRG includes rules that reflect the full repertoire of CHR, including guards, pragma declarations, etc. Among other features, it includes context-sensitive rules that may refer to arbitrary symbols to the left and right of the sequence recognized as a specific phrase. The following excerpt of a CHRG for simple coordinating sentences such as *"Peter likes and Mary detests spinach"* illustrates CHRG's propagation grammar rules and also the use of syntactic context.

```
subj(A), verb(V), obj(B) ::> sent(s(A,V,B)).
subj(A), verb(V) /- [and], sent(s(_,_,B)) ::> sent(s(A,V,B)).
```

The first rule is to be understood in the usual way that a complete `sub-verb-obj` sequence can be reduced to a `sent` node. The second rule is an example of a context-sensitive rule: It applies to a `subj-verb` sequence only if followed by terminal symbol "`and`" and another `sent` node, and in this case the incomplete sentence takes its subject, matched by variable `B`, from this following `sent` node. The marker "`/-`" separates the `subj-verb` sequence from the required right context; a similar marker may indicate left context. The grammar rules above are compiled into the following CHR rules; variables `X0`, `X1`, etc. stand for word boundaries.

```
subj(N0,N1,A), verb(N1,N2,V), obj(N2,N3,B) ==> sent(N0,N3,s(A,V,B)).
subj(N0,N1,A), verb(N1,N2,V), token(N2,N3,and), sent(N3,N4,s(_,_,B)) ==> sent(N0,N2,s(A,V,B)).
```

A simplification rule of CHRG (indicated by an arrow "`<:>`") is compiled in a similar way into a CHR simplification rule; however, if context parts are present, the result is a CHR simpagation rule that avoids the removal of nodes matched by the context. The system may, optionally, be instructed to optimize the translation by adding passive pragmas; we ignore this in the presentation.

3 Abduction in CHR Grammars and the motivation for introducing multiple constraint stores

A grammar rule may also refer to constraint symbols that are not treated as grammar symbols. The following artificial rule indicates the principle.

```
[bah], n(X), {hyp(X,Y)} ::> m(Y), {hyp(Y,X)}.
```

It indicates that token "bah" followed by a grammar symbol n(X) gives rise to the recognition of the symbol m(Y) *provided* that a constraint hyp(X,Y) is present in the constraint store, and in which case a new constraint hyp(Y,X) is added to the constraint store. A grammar can also include CHR rules that apply to such constraints; in the following we refer to such rules as *integrity constraints* and the indicated CHR constraints as *abducibles.*

Abduction in language interpretation means to hypothesize possible facts about the semantic context that make it possible to explain why a certain utterance can have been honestly given. The following rule indicates that if the hypothesis have_problem holds, then an utterance "help" is meaningful as an exclamation.

```
[help], {have_problem} ::> exclamation.
```

Written in this way, the rule is useful only when all contextual facts are known in advance, but it could in principle be run through an abductive interpreter to produce such facts when needed. In CHRG we can achieve this effect simply by moving the reference to these facts to the other side of the implication as follows:

```
[help] ::> exclamation, {have_problem}.
```

Intuitively it reads: If token "help" is observed, it is feasible to assert "have_problem" and, thus, under this assumption conclude exclamation.

The creation of a new abducible, being added to the constraint store, may trigger integrity constraints that refine the set of abducibles or reject an incompatible set. A formalization and explanation why this trick works and a comprehensive example can be found in [3, 5].

This principle only works correctly when there is no ambiguity involved. Special action needs to be taken to avoid confusion in the following cases:

- When two different interpretations of a text are possible, one in which a specific token, say "green", is read as an adjective standing for a colour and another one in which it is read as the noun synonymous with "lawn"; a single set of abducibles recording both is meaningless.
- When an abducible includes a variable as in he(X), where X is supposed to match an individual, and both X=peter and X=paul are feasible assignments, both of which should be tried out.
- In case an integrity constraint identifies a contradiction, e.g., day, night ==> fail, it causes the entire computation to fail in a committed choice language such as CHR; this is wrong as it prevents other alternatives to be tried out.

Some, but not all, of these aspects can be handled by the introduction of backtracking, but we do not consider this a viable alternative as backtracking may lead to a combinatorial explosion due the same choices being undone and redone over and over again (a phenomenon that can be experienced with DCGs). A way to approach these problems is to call for multiple constraint stores, one for each (partial) interpretation under consideration. This is not supported by CHR, but it can be simulated by a straightforward indexing technique that we have used in a prototype, written in the SICStus Prolog version of CHR [12]. Each grammar node has a unique index (actually a Prolog variable) that appears as an extra argument which also is attached to each abducible related to it. Each time a grammar rule is applied to a sequence of nodes (matched by the head of the rule), copies with a new index are made of all abducibles associated with these nodes (plus any new abducibles introduced by the given rule). Integrity constraints are modified so they apply only for abducibles with identical index variable, e.g., f(F1,C)/f(F2,C)==>F1=F2 is translated into f(I,F1,C)/f(I,F2,C)==>F1=F2.

This approach is of course very inefficient as the copying requires a heavy computational overhead when programmed in Prolog, and it is unavoidable that propagation integrity constraints are applied to the copy constraints that have already been applied once to the originals. A special control structure is compiled into the body of the grammar rules so that a failure produced by an integrity constraint is caught and converted to a silent removal of the given node rather than leading to the termination of the entire computation. In a grammar with context parts, each rule needs to be equipped with a guard to prevent confusion of different interpretations of overlapping substrings; this is a bit complicated to explain and is ignored in the following.

4 A tailored architecture with multiple constraint stores for abductive CHRGs

Basically a model is called for in which each grammar node has its own constraint store. Instead of (ab)using CHR as indicated in the previous, we propose a new implementation in which the underlying data structure supports the maintenance of multiple and overlapping constraint stores in an efficient way.

There are basically two ways to implement such a structure of overlapping local constraint stores, by *sharing* or by *copying*; a choice needs to be made although hybrid solutions also are possible. A shared representation indicates that each store is represented by a set of pointers to included substores plus a local component recording any new constraint included at this level, bindings to variables (at any level as sideeffects on lower levels will introduce confusion), and finally a table of those constraints below considered to be deleted (by a simplification rule). Structure sharing is often preferred in order to save space, but in this case it seems to create long chains so that the matching of head patterns with the constraint store slows down, not to mention the execution of a unification.

Instead we propose a model based on efficient copying of local constraint stores. In order for this to be feasible, we have given priority to the following properties:

– A compact representation of terms so that only small amounts of data needs to be copied.
– A representation that avoids the generation of chains of variable references whenever possible.
– Relocation from one position in the RAM store to another should be done without changing pointers.

In the following we sketch the representation that used in a prototype under development.

4.1 Overall structure

The execution pattern is indicated by the following abstract interpreter; for simplicity we consider only propagation rules without guards and built-in's.

A *multi constraint store* is a set of pairs $\langle G, S\rangle$ where G is a grammar symbol (with phrase boundaries) and S a constraint store consisting of abducible atoms only. A *derivation* is a sequence of multi constraint stores $M_1, \ldots, M_n$, with M_n called a *final* state, such that:

– M_1 has one component for each input token as indicated by this example:
  $M_1 = \{\langle \texttt{token(0,1,the)}, \emptyset\rangle, \langle \texttt{token(1,2,man)}, \emptyset\rangle, \langle \texttt{token(2,3, walked)}, \emptyset\rangle\}$
– M_{i+1} is derived from M_i by the application of a rule in one of the following ways:
  • There is an instance of a grammar rule $G_1, \ldots, G_m \texttt{::>} G_0, \{A\}$ with $\langle G_j, S_j\rangle \in M_i$ for $j = 1 \ldots m$; A is a set of abducibles. Then
  $M_{i+1} = M_i \cup \{\langle G_0, S_1 \cup \cdots \cup S_m \cup A\rangle\}$
  • An integrity constraint can apply to one of the local constraint stores S in M_i producing a state S' and possible sideeffects on variables in G given by substitution σ. Then
  $M_{i+1} = M_i \setminus \{\langle G, S\rangle\} \cup \{\langle G\sigma, S'\rangle\}$.
  However, if S' is failed, $M_{i+1} = M_i \setminus \{\langle G, S\rangle\}$.
– No rule is applied twice to the same constraints.
– No rule can apply to M_n.

The final state, then, contains all possible interpretations of the input text, so for example two alternative `sentence` nodes would appear for a sentence that can be understood in two different way.

The following detailed computation rule is imposed:

– Components of the initial store are entered one by one from left to right, each at a point when no rule can apply.
– Whenever a grammar rule is applied, integrity constraints apply as long as possible before another grammar rule can apply.

4.2 Representation of terms

We consider a system that is liberated from Prolog so that the representation of terms does not need to support backtracking: When a failure in a branch of computation appears, its local constraint store can be discarded immediately in one piece.

The key to achieve efficient copying is to use relative addresses so that a pointer is given as a number that indicates how many RAM cells away the indicated item can be found. We illustrate the approach by a small example showing part of a constraint store that holds three terms `p(X)`, `q(Y)`, `r(Z)`, initially with all variables unbound. We can indicate this by three consecutive records as follows:

p	variable offset=0
q	variable offset=0
r	variable offset=0

The zero offset indicates that the given variables are located in the record for each term. The unification `X=Y` aliasing the two variables modifies the store into the following.

p	variable offset=1
q	variable offset=0
r	variable offset=0

The `offset=1` indicates that the variable is stored one record below (in practice we would count the distance in number of bytes as records may vary in length). The unifications `X=a`, `Z=Y` lead to the following picture:

`p`	`variable offset=1`
`q`	`constant=a`
`r`	`variable offset=-1`

The indicated representation has the great advantage that its interpretation is independent of the actual position in RAM; copying it in one piece to another location does not affect its integrity. Furthermore, the lengths of variable reference chains need never be greater than one.

This model can be extended for complex structures in a standard way, and matching (CHR's principle for identifying constraints for the application of a rule) as well as unification can be implemented in straightforward ways.

The details have not been worked out yet, but it appears that the union of different constraint stores can be produced basically by copying their different data areas one by one into a new consecutive area.

Deletion of constraints (by a simplification or simpagation rule) can be implemented by adding an extra bit to each record.

5 Extensions and optimizations

5.1 Incremental integration of local stores and integrity checking

Abductive language interpretation is likely to produce a lot of intermediate candidate interpretations that are anyhow ruled out by integrity constraints. In fact, we consider it highly important, as part of preparing an abductive grammar, carefully to prepare a system of integrity constraints that catches nonsense states as early as possible; this is necessary in order to keep down the explosive complexity of abduction.

We will consider it as the rule rather that the exception that a newly constructed component $\langle G, S\rangle$ is discarded due to failure of integrity constraints applied to S. Inspired by our own work on efficient integrity checking in data integration systems [7, 8] we propose to optimize this step in the following way.

Call the set of integrity constraints Γ and assume that the application of a grammar rule involves the construction of a new set of abducibles $S_{\text{new}} = S_1 \cup \cdots \cup S_m \cup A$; we can assume that each S_i satisfies Γ and the task is to find whether this is also the case for S_{new}. We understand here a set of integrity constraints as a function from one or more constraint stores to a new constraint stores or *failure*.

Whenever S and S' are two constraint store, both known to satisfy Γ, the methods of [8] can produce an optimized version Γ' for the specific problem of determining the result of $\Gamma(S \cup S')$; we write its application $\Gamma'(S, S')$. Roughly stated, Γ' applies rules only to combinations of constraints coming from both S and S' (combinations exclusively in S or exclusively in S' have been tested at earlier stages). Let in a similar way $\Gamma''(S, S')$ represent another optimized version that assumes consistency of S but not necessarily of S'. (In fact, [8] considers only integrity constraints that are pure testers, so the methods need to be adapted to CHR rules that are "active" in the sense that they may add extra abducibles, recursively triggering other rules.)

The following algorithm shows how S_{new} can be constructed in an incremental and optimal way from the component stores.

```
1. $i := 1$; $S_{\text{new}} := S_1$;
2. $S_{\text{new}} := \Gamma'(S_{\text{new}}, S_i)$;
3. if $S_{\text{new}} = \textit{failure}$ then failure-exit;
4. if $i < m$ then $\{i := i + 1$; go to 2$\}$;
5. $S_{\text{new}} := \Gamma''(S_{\text{new}}, A)$;
6. if $S_{\text{new}} = \textit{failure}$ then failure-exit else normal-exit;
```

5.2 Explicit state splitting

The proposed architecture can be extended to handle a *split* operator which can be used in rule bodies; it is similar to disjunction in CHR$^{\vee}$ considered by [1, 2]. We denote it here by an infix symbol "`or`". If, for example, c_1 `or` c_2 is executed, it means that the given component (grammar symbol with local constraint store) immediately splits into two, one including c_1 and another one including c_2, and execution continues separately in each of the two components. With our state model it is obvious to have c_1 continue in its current component after a clone has been produced for the processing of c_2.

As shown by [2], it is possible to represent any Prolog program as a set of simplification rules with splits in the body, one alternative for each defining clause. First of all, this shows that a variant without grammar symbols of our architecture can provide an efficient implementation of CHR$^{\vee}$.

Secondly it allows to extend our system with a common principle for obtaining minimality of abductive answers (measured in the number of literals), which is applied in many abductive systems, e.g., [10]. Whenever two abducibles $a(s)$ and $a(t)$ occur simultaneously, the interpreter tries to unify s and t, and, if this leads to a failure, the alternative $s \neq t$ is tried out, typically under backtracking. This principle can be implemented in our system by adding, for each abducible predicate, a rule such as `a(X)/a(Y) <=> X=Y or dif(X,Y).`

However, we cannot recommend this principle as a standard for abduction as it it implies exponentially many combinations to be tried out. We do not go into a detailed discussion here, but we see it as a symptom of a badly specified abductive problem if this principle is essential for keeping the number of abduced atoms small: carefully designed integrity constraint that describe essential domain properties need to be added to the specification. We believe that an explicit split operator is a very useful device in the development of abduction based language interpretation systems. It allows integrity constraints and grammar rules explicitly to state that alternative choices should be tried.

6 Summary and related work

We have proposed a new implementation of CHR Grammars which is intended to make abductive language interpretation possible in an efficient way. Together with the high flexibility and expressibility in this formalism, including integrity constraints written as CHR rules and a new splitting operator, we hope to provide a setting that makes abduction more attractive and realistic for language processing.

The approach to abduction taken here appears to be considerably more efficient that other known approaches; the price to be payed, however, is a limitation on the use of negation. Only so-called explicit negation simulated by integrity constraints is possible at present.

A prototype implementation is currently under development and benchmark tests will be made in the future in order to see whether our expectations about efficiency are fulfilled.

We still need to compare the proposed implementation architecture with other committed choice logic programming languages such as Parlog [9], Guarded Horn Clauses [13], and Concurrent Prolog [11].

Acknowledgement
Thanks to Matthieu Le Brun who is developing the prototype implementation. This research is supported in part by CONTROL project, funded by the Danish Natural Science Research Council, and also so in part by the IT University of Copenhagen.

References

1. Abdennadher, S. & Christiansen, H. (2000) An Experimental CLP Platform for Integrity Constraints and Abduction. *Proceedings of FQAS2000, Flexible Query Answering Systems: Advances in Soft Computing series*, pp. 141–152. Physica-Verlag (Springer).
2. Abdennadher, S. & Schütz, H. (1998) CHR$^\vee$: A flexible query language. *Proceedings of FQAS '98: Lecture Notes in Computer Science 1495*, pp. 1–14. Springer-Verlag.
3. Christiansen, H., (2002) Abductive language interpretation as bottom-up deduction. *Proc. of NLULP 2002, Natural Language Understanding and Logic Programming, Datalogiske Skrifter* 92, pp. 33–48 Roskilde University, Denmark.
4. Christiansen, H., *CHR Grammar web site*, Released 2002. http://www.dat.ruc.dk/~henning/chrg/
5. Christiansen, H. CHR grammars. To appear in *Theory and Practice of Logic Programming*, special issue on Constraint Handling Rules, expected publication date medio 2005.
6. Christiansen, H., A constraint-based bottom-up counterpart to DCG. *Proceedings of RANLP 2003, Recent Advances in Natural Language Processing*, 10–12 September 2003, Borovets, Bulgaria. pp. 105–111.
7. Christiansen, H., Martinenghi, D., Simplification of database integrity constraints revisited: A transformational approach. To appear in Proceedings of LOPSTR 2003, International Symposium on Logic-based Program Synthesis and Transformation, Uppsala, Sweden, August 25–27, 2003. *Lecture Notes in Computer Science*, 2004.
8. Christiansen, H., Martinenghi, D., Simplification of integrity constraints for data integration, *Third International Symposium on Foundations of Information and Knowledge Systems (FoIKS), February 17–20, 2004 — Vienna, Austria.* Lecture Notes in Computer Science 2942, pp. 31–48, 2004.
9. Clark, K., Gregory, S. (1986) PARLOG: parallel programming in logic, *ACM Trans. Program. Lang. Syst.*, vol 8, no. 1, pp. 1–49.
10. Kakas, A. C., Michael, A., Mourlas, C. (2000) ACLP: Abductive Constraint Logic Programming. *Journal of Logic Programming*, Vol. 44 (1–3), pp. 129–177.
11. Shapiro, E.Y., Takeuchi, A. (1983) Object Oriented Programming in Concurrent Prolog. *New Generation Computing* 1(1), pp. 25–48.
12. *SICStus Prolog user's manual.* Swedish Institute of Computer Science, 2004. Most recent version available at `http://www.sics.se/isl`.
13. Ueda, K. (1988) Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. *Programming of Future Generation Computers*, Nivat, M. and Fuchi, K. (eds.), North-Holland, Amsterdam, pp.441–456.