First Workshop on Constraint Handling Rules

Thom Frühwirth, Marc Meister (eds.)
University of Ulm

Abstract

The Constraint Handling Rules (CHR) language has become a major specification and implementation language for constraint-based algorithms and applications. Algorithms are often specified using inference rules, rewrite rules, sequents, proof rules, or logical axioms that can be directly written in CHR. Based on first order predicate logic, this clean semantics of CHR facilitates non-trivial program analysis and transformation.

Several implementations of CHR exist in Prolog, Haskell, and Java. A particular emphasis of this first workshop was the comparison, joint development, consolidation and common extension of the various CHR implementations.

The workshop was held May 10–14, 2004 at the University of Ulm. Three papers were selected for this technical report. Slides of the seven presentations are available online.

http://www.informatik.uni-ulm.de/pm/veranstaltungen/chr2004/

Selected Papers

Henning Christiansen
CHR Grammars with multiple constraint store

Tom Schrijvers and Bart Demoen
The K.U.Leuven CHR system: implementation and application

Peter J. Stuckey, Martin Sulzmann and Jeremy Wazny
The Chameleon System

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:
  $M_1 = \{\langle$`token(0,1,the)`$, \emptyset\rangle, \langle$`token(1,2,man)`$, \emptyset\rangle, \langle$`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 $G_1, \ldots, G_m$ `::>` $G_0, \{A\}$ 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 $S_{\text{new}} =$ *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 $S_{\text{new}} =$ *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.

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^{i}PS^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/.

The Chameleon System

Peter J. Stuckey *, Martin Sulzmann †and Jeremy Wazny ‡

April 22, 2004

Abstract

We give an overview of the major features of the Chameleon programming language, as well as use of the system, from a Haskell programmer's point of view. Chameleon supports type-class overloading a la Haskell. The novelty of Chameleon is to allow the programmer to specify almost arbitrary conditions on type classes in terms of Constraint Handling Rules (CHRs). In fact, Chameleon's entire type inference system and evidence translation of programs is phrased in terms of CHRs.

1 Introduction

Chameleon is a Haskell-style language to experiment with advanced type extensions. Chameleon covers most of the Haskell syntax [Has]. In contrast to Haskell, we only support singly-overloaded methods.

Example 1 *For instance, consider the following Haskell 98 type class and instance declarations.*

```
class Eq a where (==) :: a->a->Bool
instance Eq a => Eq [a] where
 (==) [] [] = True
 (==) (x:xs) (y:ys) = ((==) x y) && ((==) xs ys)
 (==) _ _ = False
```

The above example is written as follows in Chameleon.

Example 2

```
overload eqC

instance EqC (a->a->Bool) => EqC ([a]->[a]->Bool) where
 eqC [] [] = True
 eqC (x:xs) (y:ys) = (eqC x y) && (eqC xs ys)
 eqC _ _ = False

rule EqC a ==> a=t->t->Bool
```

*Dept. of Computer Science and Software Engineering, The University of Melbourne, Parkville 3052, Australia. pjs@cs.mu.oz.au

†School of Computing, National University of Singapore, S16 Level 5, 3 Science Drive 2, Singapore 117543 sulzmann@comp.nus.edu.sg

‡Dept. of Computer Science and Software Engineering, The University of Melbourne, Parkville 3052, Australia. jeremyrw@cs.mu.oz.au

In Chameleon each constraint `Method t` refers to an overloaded use of identifier `method` at type `t`. Hence, the statement `overload eqC` is similar to `class EqC a where eqC :: a` in Haskell. The novelty of Chameleon is to write almost arbitrary programs (on the level of types) in terms Constraint Handling Rules (CHRs) [Frü95]. The Chameleon programmer can introduce such CHRs via the `rule` keyword. For example, the above CHR `rule EqC a ==> a=t->t->Bool` (in combination with `overload eqC`) essentially mimics the Haskell class declaration `class Eq a where (==) :: a->a->Bool`. Note that in Chameleon we do not impose any conditions on type class constraints appearing in the instance context. E.g. Haskell 98 demands that all type class parameters must be variables.

In the following, we present an overview of the major features of the Chameleon programming language, as well as use of the system. In Section 2 we give an overview of the kind of CHRs found in Chameleon. In Section 3 we consider type inference whereas in Section 4 we consider translating Chameleon, i.e. replacing uses of overloaded methods by some specific instances. In Section 5 we consider some extended examples which make use of Chameleon's CHR-based type programming capabilities. In Section 6 we show how to make use of existing Haskell code within Chameleon. Finally, in Section 7 we show how to use the system to execute Chameleon programs.

The latest Chameleon distribution is available via

`http://www.comp.nus.edu.sg/~sulzmann/chameleon`

2 Constraint Handling Rules

In general, type inference works by *generating* and *solving* constraints. Here, constraint solving is performed w.r.t. the relations imposed on overloaded identifiers which are specified in terms of CHRs. For example, in Haskell the declaration `class Eq a where (==) :: a->a->Bool` states that any use of `(==)` must be of shape `t->t->Bool` where the constraint `Eq t` refers to a particular instance of the equality type class. In Chameleon, we can simply write

```
rule EqC a ==> a=t->t->Bool
```

to achieve the same behavior (keep in mind that constraints in Chameleon refer to exactly one method). Via the `rule` keyword the Chameleon programmer can introduce the following two forms of CHRs.

```
rule Method1 t1, ..., Methodn tn ==> s=t
```

and

```
rule Method1 t1, ..., Methodn tn ==> Method t
```

Both kinds are referred to as CHR *propagation* rules. The first kind allows us to impose stronger conditions on the shape of types. The second kind of CHRs is useful to impose dependencies among overloaded instances.

Example 3 *For instance, consider the following Chameleon program.*

```
overload ordC
rule OrdC a ==> EqC a
rule OrdC a ==> a=t->t->Bool
instance OrdC (a->a->Bool) => OrdC ([a]->[a]->Bool)
```

Note that if the instance body is omitted we assume the instance definition is undefined. The constraint symbol `OrdC` refers to an ordering relation. Hence, it is natural to require that each such instance implies an equality instance `EqC`. Indeed the above CHRs are pretty much equivalent to `class Eq a => Ord a where (<) :: a->a->Bool` in Haskell.

2.1 Confluence

In Chameleon, CHRs are checked for "consistency" before we proceed with type inference.

Example 4 *Consider the following Chameleon example where we have dropped* `OrdC (a->a->Bool)` *from the instance context.*

```
overload eqC
overload ordC

instance EqC (a->a->Bool) => EqC ([a]->[a]->Bool) where  -- (E1)
 eqC [] [] = True
 eqC (x:xs) (y:ys) = (eqC x y) && (eqC xs ys)
 eqC _ _ = False

rule EqC a ==> a=t->t->Bool

instance {-OrdC (a->a->Bool)=>-} OrdC ([a]->[a]->Bool)   -- (O1)

rule OrdC a ==> a=t->t->Bool

rule OrdC a ==> EqC a                                    -- (E2)
```

Chameleon complains that CHRs are "non-confluent". Note that each `instance C => U t` implies a (single-headed) *simplification* CHR of the form `rule U t <==> C` (the programmer does not have to specify such simplification rules explicitly, they are automatically generated from instances). We generally require that CHRs are *confluent*, i.e. the order of CHR applications does not matter. This is not the case for the above example. In case we encounter `Ord([a]->[a]->Bool)`, we can either apply rule (O1) which yields the empty constraint represented by `True`. Alternatively, we can apply rule (E2) which yields `Ord([a]->[a]->Bool),Eq ([a]->[a]->Bool)`. Applications of rules (E1) and (O1) finally leads to `Eq (a->a->Bool)` which is different from `True`. Hence, CHRs are non-confluent. In fact, a similar check is enforced in Haskell, however, the notion of confluence is more general. Note that if one may wish, the confluence check can be switched off via the "–no-conf" option.

2.2 Termination

For terminating CHRs, we can test for confluence by checking all "critical pairs". In fact, if the confluence check in Chameleon is invoked we assume that CHRs are terminating. However, we can easily write a "non-terminating" Chameleon program.

```
instance Erk [a] => Erk a
```

The CHR implied by the above instance is non-terminating. By default Chameleon performs a simple termination check. Note that CHRs are a Turing-complete language. Hence, any termination check must be necessarily incomplete. Via the "–no-term" option, the user can switch off the termination check.

2.3 Range-Restriction

Consider

```
instance C (Int->Int->Int)
instance C (a->b->c) => D ([a]->[b]) -- (D1)
```

Chameleon complains that the CHR resulting from (D1) is not "range-restricted". Note that if we ground variables `a` and `b` we still find the non-ground term `c` in the instance context. Range-restrictedness ensures that there are no unconstrained variables which potentially endangers completeness of type inference. In the upcoming Section 5.2 we consider an instance (Select) which is non-rangestricted but safe! Via the "–no-rr" option, the user may switch off the check for range-restriction.

2.4 CHR Solver

A description of the internal details of the CHR solver are beyond the scope of this document. Note that Chameleon comes with a fully-fledged CHR solver (on the level of types). That is, we also support *multi-headed* simplification rules.

Example 5 *In the following Chameleon program we model stack operations (on the level of types) such as push and pop. Their meaning is defined in terms of CHRs.*

```
interface Prelude -- will be explained in detail later
                  -- in essence, we teach Chameleon how to use the Haskell 98 Prelude

-- we model the stack as a singleton list
data Nil = Nil
data Cons x xs = Cons x xs

-- some stack elements
data V1 = V1
data V2 = V2
data V3 = V3

overload push
overload pop
overload stack

-- two multi-headed simplification rules
rule Push (x), Stack (s)      <==> Stack (Cons x s)
rule Pop (), Stack (Cons x s) <==> Stack (s)
rule Pop (), Stack (s)         ==> s=Cons x s'

-- we introduce a CHR to model initial constraint store
overload initStore
rule InitStore a <==> Stack (Nil), Push (V1), Push (V2)

run = initStore
```

Obviously, we cannot run (on the level of values) the above program. But we can execute the CHR program by invoking Chameleon's CHR-based type inference mechanism. For example, the call `chameleon --no-term -d stack.ch` invokes Chameleon in pure type inference mode (-d) where we have switched of our CHR termination check (–no-term). We reach a command-line environment with similar functionality compared to HUGS [HUG] or GHCi (besides executing program code) [GHC]. For example,

```
stack.ch> :t run
(Stack(Cons V2 (Cons V1 Nil))) => a
```

allows us to execute the above CHR program. The constraint component of the above type holds the result of running the above CHR program on the given constraint store.

3 Type Inference

Type inference in Chameleon proceeds by generating the appropriate constraints out of the program text and solving them w.r.t. the given set of CHRs. A program is type correct if there are no inconsistent constraints and all user-provided type annotations are correct. Additionally, we also check that expressions are unambiguous (though the user can switch off this check via the `--no-unambig` option). In case of a type error, incorrect type annotation or ambiguous expression Chameleon will issue an error message. The user can then invoke the Chameleon Type Debugger [SSW03b, SSW03a] to get some assistance in fixing such errors.

In contrast to other Haskell implementations such as Hugs and GHC we do not pretty-print types. Consider

Example 6

```
interface Prelude

overload eqC
overload ordC

rule EqC a ==> a=t->t->Bool
rule OrdC a ==> a=t->t->Bool
rule OrdC a ==> EqC a

f = ordC
```

If we invoke Chameleon's type inference/debugging environment (via `chameleon -d eq2C.ch`) and query the type of `f` we find the following

```
eq2C.ch> :t f
(EqC(a -> a -> Bool),OrdC(a -> a -> Bool)) => a -> a -> Bool
```

The constraint `EqC(a -> a -> Bool)` might be considered redundant (but harmless). Indeed, `OrdC(a -> a -> Bool) => a -> a -> Bool` is equivalent to the above type. We plan to incorporate pretty-printing of types in future versions of Chameleon.

In our current implementation type inference is more "lazy" compared to the HUGS or GHC.

Example 7 *Consider*

```
interface Prelude

f x = let h = (x::Int)
      in x
```

Chameleon reports that `f` has type `a->a` whereas e.g. Hugs reports `Int->Int`. Note that `h` only constraints the type of `x` without actually being used in `f`'s function body. Hence, Chameleon neglects the constraints associated to `h`. Obviously, we could easily enforce "strict" type inference but we have not done so in our current implementation.

Note also that Chameleon does not enforce the Haskell monomorphism restriction. Furthermore, there is no defaulting in Chameleon. For example, the following program is accepted by Chameleon, while Haskell would reject it.

```
p :: (Int, Float)
p = let n = 1
    in  (n,n)
```

The monomorphism restriction and defaulting often eliminate bothersome sources of ambiguity in Haskell programs. Consequently, there are ambiguous Chameleon programs whose Haskell counterparts are unambiguous. Consider the following:

```
one = if 1 == 1 then 1
                else undefined
```

Chameleon will report this as ambiguous, whereas Haskell will accept it.

It has been observed that all known Haskell implementations reject some seemingly well-typed programs.

Example 8 *Here is a recast of a Haskell program published on the Haskell-Cafe [JN00] mailing list in terms of Chameleon.*

```
interface Prelude

-- we model
-- class C t where op :: t->Bool
-- instance C [a]
hconstraint C
extern op :: C t => t->Bool
rule C [a] <==> True

p y = (let f :: c->Bool
           f x = op (y >> return x)
       in f,  y ++ [])

q y = (y ++ [],
       let f :: c->Bool
           f x = op (y >> return x)
       in f)
```

The above program is type correct. Note that Hugs and GHC either reject `p` or `q` whereas Chameleon accepts both functions!

In a recent paper [SSW04] we elaborate on this topic in detail.

Example 9 *Here is another interesting example from this paper*

```
interface Prelude

-- class HasEmpty a where isEmpty :: a->Bool
hconstraint HasEmpty
extern isEmpty :: HasEmpty a => a->Bool

hconstraint Erk

{-
instance HasEmpty [a]
instance HasEmpty (Maybe a)
```

```
instance Erk m => HasEmpty (m a)
instance Erk []
instance Erk Maybe
-}

rule HasEmpty [a] <==> True       -- (H1)
rule HasEmpty (Maybe a) <==> True -- (H2)
rule HasEmpty (m a) <==> Erk m    -- (E1)
rule Erk [] <==> True             -- (E2)
rule Erk Maybe <==> True          -- (E3)

test :: (Monad m, HasEmpty (m (m a))) => m a -> Bool
test y = let f :: d->Bool
             f x = isEmpty (y >> return x)
         in f y
```

We can invoke Chameleon via `chameleon --no-conf -d hasempty.ch` to check that the above program is type correct. We need to switch off the confluence check because the above instances are overlapping. For details we refer to the above mentioned paper.

4 Translating Chameleon

To resolve overloading on the value level we apply the well-known "evidence" translation scheme. Chameleon is translated to "plain" Haskell. However, there are some subtle differences among Chameleon-style and Haskell-style overloading.

Example 10 *Consider a variation of Example 4. Note that instance (O1) includes context* `OrdC (a->a->Bool)`. *Hence, CHRs are confluent.*

```
overload eqC
overload ordC

instance EqC (a->a->Bool) => EqC ([a]->[a]->Bool) where
 eqC [] [] = True
 eqC (x:xs) (y:ys) = (eqC x y) && (eqC xs ys)
 eqC _ _ = False

rule EqC a ==> a=t->t->Bool

instance OrdC (a->a->Bool) => OrdC ([a]->[a]->Bool)  -- (O1)

rule OrdC a ==> a=t->t->Bool

rule OrdC a ==> EqC a

f :: OrdC (a->a->Bool) => a->a->Bool           -- incorrect type annotation!
f = eqC
```

The above looks like a perfectly reasonable program. Surprisingly, Chameleon rejects the above program because `f`'s type annotation is incorrect. Note that in Haskell `Ord a` refers to a type class which contains a method `(<)` of type `a->a->Bool` and (due to sub-classing) also refers to

an equality method `(==)` of type `a->a->Bool`. However, in Chameleon, each constraint `Method t` refers exactly to an overloaded identifier `method` of type `t`. Hence, the statement `rule OrdC a ==> EqC a` has only a meaning on the level of types, i.e. for each `OrdC t` there must also be a `EqC t`. In order to resolve overloading on the value level, we need to explicitly mention a constraint `Method t` for each use of `method` at type `t`.

Example 11 *Consider the following variation of Example 10. In the function body of* `f` *we find now* `ordC` *instead of* `eqC`.

```
interface Prelude -- will be explained in detail later
                  -- in essence, we teach Chameleon how to use the Haskell 98 Prelude

overload eqC
overload ordC

instance EqC (a->a->Bool) => EqC ([a]->[a]->Bool) where
 eqC [] [] = True
 eqC (x:xs) (y:ys) = (eqC x y) && (eqC xs ys)
 eqC _ _ = False

rule EqC a ==> a=t->t->Bool

instance OrdC (a->a->Bool)=> OrdC ([a]->[a]->Bool) where
 ordC = undefined
-- if the instance body is left empty we assume the "undefined" instance body by default

rule OrdC a ==> a=t->t->Bool

rule OrdC a ==> EqC a

f :: OrdC (a->a->Bool) => [a]->[a]->Bool
f = ordC
```

The translation yields the following result (call `chameleon -o eqordC.hs eqordC.ch`).

```
import Prelude

ec3 :: ((->) (((->) (a)) (((->) (a)) (Bool)))) (((->) ([(a)])) (((->) ([(a)])) (Bool)))
ec3 (pEqCIIARR_v18026I_IIARR_v18026I_BoolII) ([]) ([]) = let {pEqCIIARR_a18373I_IIARR_a18373I_BoolII
= pEqCIIARR_v18026I_IIARR_v18026I_BoolII} in  (True)
ec3 (pEqCIIARR_v18026I_IIARR_v18026I_BoolII) ((:) (x) (xs)) ((:) (y) (ys)) = let {pEqCIIARR_a18373I_II
ARR_a18373I_BoolII  = pEqCIIARR_v18026I_IIARR_v18026I_BoolII} in  ((&&) (((pEqCIIARR_a18373I_IIARR_a18
373I_BoolII) (x)) (y)) ((((ec3) (pEqCIIARR_a18373I_IIARR_a18373I_BoolII)) (xs)) (ys)))
ec3 (pEqCIIARR_v18026I_IIARR_v18026I_BoolII) (_) (_) = let {pEqCIIARR_a18373I_IIARR_a18373I_BoolII  =
pEqCIIARR_v18026I_IIARR_v18026I_BoolII} in  (False)

ec83 :: ((->) (((->) (a)) (((->) (a)) (Bool)))) (((->) ([(a)])) (((->) ([(a)])) (Bool)))
ec83 (pOrdCIIARR_v18174I_IIARR_v18174I_BoolII) = let {pOrdCIIARR_a18427I_IIARR_a18427I_BoolII  = pOrdC
IIARR_v18174I_IIARR_v18174I_BoolII} in  (undefined)

f :: ((->) (((->) (a)) (((->) (a)) (Bool)))) (((->) ([(a)])) (((->) ([(a)])) (Bool)))
f (pOrdCIIARR_v18236I_IIARR_v18236I_BoolII) = let {pOrdCIIARR_a18542I_IIARR_a18542I_BoolII  = pOrdCIIA
RR_v18236I_IIARR_v18236I_BoolII} in  ((ec83) (pOrdCIIARR_a18542I_IIARR_a18542I_BoolII))
```

which can be executed by any Haskell compiler. Briefly, we turn instance declarations into function definitions. E.g. `ec3` corresponds to `instance EqC (a->a->Bool) => EqC ([a]->[a]->Bool)`. Note the correspondence between `ec3`'s (pretty-printed) type annotation `(a->a->Bool)->[a]->[a]->Bool` and the instance declaration.

Obviously, the Chameleon user does not need to understand the the details of the implementation scheme behind Chameleon. The curious reader is referred to [SW] for more details. Note in our latest release we have incorporated the *weak* entailment check discussed on page 6 in [SW]. .

5 Examples

The real fun of programming in Chameleon starts when writing programs on the level of types. We briefly illustrate Chameleon's type level programming ability in connection with "functional dependencies". Functional dependencies [Jon00] are a popular type class extension where the programmer can take (to some extent) control over the type inference process. Consider the following Haskell program.

```
class C a b | a->b where c :: a->b
instance C Int Bool
```

The functional dependency `| a->b` states that in `C a b` the first component uniquely determines the second component. For example, if type inference encounters `C a b` and `C a c`, types are *improved* by adding the equality constraint `b=c`. In case of `C Int b` the functional dependency in combination with above the instance implies that `b=Bool`. In Chameleon, we can make such *improvement rules* explicit in terms of CHRs. Here is the Chameleon equivalent of the above program.

```
overload c
rule C x ==> x=a->b
-- FD rule
rule C (a->b), C (a->c) ==> b=c
instance C (Int->Bool)
-- instance improvement rule
rule C (Int->b) ==> b=Bool
```

In fact, CHRs not only give us a sound foundation for functional dependencies, CHRs even allow us to go beyond functional dependencies. The interested reader is referred to [GJDS04] for more details. Here, we briefly present two examples making use of more powerful functional dependencies expressible.

5.1 A Generic Family of Zip Functions

We start off with an encoding in Haskell, point out some limitations of functional dependencies and then present a solution in Chameleon.

```
-- Haskell encoding
  zip2 :: [a]->[b]->[(a,b)]
  zip2 (a:as) (b:bs) = (a,b) : (zip2 as bs)
  zip2 _      _      = []

  class Zip a b c | a c -> b, b c -> a where
    zip :: [a] -> [b] -> c
```

```
  instance Zip a b [(a,b)] where
    zip = zip2
  instance Zip (a,b) c e => Zip a b ([c]->e) where
    zip as bs cs = zip (zip2 as bs) cs

  e2 = head (zip [True,False] ['a','b','c'])
  e3 = head (zip [True,False] ['a','b','c'] [1::Int,2])
```

Note that the inferred types of `e2` and `e3` are

```
  e2 :: Zip Bool Char [a] => a
  e3 :: Zip (Bool,Char) Int [a] => a
```

rather than

```
  e2 :: (Bool,Char)
  e3 :: ((Bool,Char),Int)
```

Indeed, the relevant improvement rule here stated in terms of a CHRs is as follows

```
rule Zip a d [(a,b)]      ==> d=b
```

To achieve the desired typing for `e2` and `e3` we would like to state a slightly stronger condition such as

```
rule Zip a d [c] ==> c=(a,b), d=b
```

However, there is no mechanism in Haskell which allows us to do so.

```
------------
-- zip.ch --
------------
-- Chameleon encoding
-- we point out parts where Chameleon allows us to impose a stronger improvement rule
interface Prelude

overload zip

zip2 :: [a] -> [b] -> [(a,b)]
zip2 (x:xs) (y:ys) = (x,y) : (zip2 xs ys)
zip2 _ _ = []

-- enforce
-- class Zip ([a]->[b]->c) where zip :: [a]->[b]->c
rule Zip x ==> x=[a]->[b]->c

-- FD
rule Zip (a->b->c), Zip (a->d->c) ==> b=d
rule Zip (a->b->c), Zip (d->b->c) ==> a=d

-- base case
instance Zip ([a]->[b]->[(a,b)])  where
   zip= zip2
```

```
-- instance-improvement
-- rule Zip ([a]->c->[(a,b)]) ==> c=[b]   -- as specified by FD
rule Zip ([a]->d->[c]) ==> d=[b],c=(a,b)  -- though we employ a slightly stronger rule
                                          -- to achieve the desired typing

rule Zip (c->[b]->[(a,b)]) ==> c=[a]

-- general case
instance Zip ([(a,b)]->[cs]->r) => Zip ([a]->[b]->[cs]->r) where
         zip as bs cs = zip (zip2 as bs) cs

-- instance improvements for this case are trivial

-- annoations are not necessary here
--e1 :: [(Bool,Char)]
e1 = head (zip [True,False] ['a','b','c'])
--e2 :: [((Bool,Char),Int)]
e2 = head (zip [True,False] ['a','b','c'] [1::Int,2])

-- to run this program
-- you need to call "chameleon -o zip.hs zip.ch"
-- you can then run zip.hs
```

Obviously, we should not add arbitrary CHRs otherwise we might endanger soundness and decidability of type inference.

5.2 Extensible Records

As another application we present an encoding of extensible records in Chameleon. The basic idea is to encode field labels and records in terms of singleton types. Note that a similar encoding with functional dependencies would be too "weak".

```
---------------
-- record.ch --
---------------
interface Prelude

-- we use singleton lists to model records

data Nil = Nil
data Cons x xs = Cons x xs

-- we use singleton numbers to model record labels

data Zero   = Zero
data Succ n = Succ n

-- result of type level computations
data T = T
data F = F
```

```
------ overloaded identifiers ------

overload eqR
overload select
overload select'
overload remove
overload remove'
overload extend
overload check
overload unique
overload unique'


------ equality among types ------

-- class declaration
rule EqR x ==> x=a->b->c

-- FD
rule EqR (a->b->c), EqR (a->b->d) ==> c=d

-- instances
instance EqR (Zero -> Zero -> T) where
         eqR Zero Zero = T
instance EqR (a->b->c) => EqR (Succ a -> Succ b -> c) where  -- (E2)
         eqR (Succ a) (Succ b) = eqR a b
instance EqR (Zero -> Succ a -> F) where
         eqR Zero (Succ a) = F
instance EqR (Succ a -> Zero -> F) where
         eqR (Succ a) Zero = F

-- instance improvement, note that for (E2) instance improvement is trivial
rule EqR (Zero->Zero->a)         ==> a=T
rule EqR (Succ a->Zero->b)       ==> b=F
rule EqR (Zero->Succ a->b)       ==> b=F


------ record selection ------

-- usage: select Record Label --> FieldEntry

-- class declaration
rule Select x ==> x=r->l->a

-- FD
rule Select (r->l->a), Select (r->l->b) ==>a=b
```

```
-- instance
instance (Select' (Cons (x1,v) e -> x2 -> b -> v'), EqR (x1->x2->b))
                => Select (Cons (x1,v) e -> x2 -> v') where
         select (Cons (x1,v) e) x2 = select' (Cons (x1,v) e) x2 (eqR x1 x2)
-- Chameleon complains that this instance is not range-restricted
-- However, we can argue that variable b (which is the trouble-maker) is
-- functionally defined by variables in the instance head

-- instance improvement is trivial here

-- class declaration
rule Select' x ==> x=e->c->a->b

-- FD
rule Select' (e->c->a->b),Select' (e->c->a->d) ==> b=d

instance Select' (Cons (x,v) e -> x -> T -> v) where
         select' (Cons (_,v) e) _ T = v
instance Select (e->x2->v') => Select' (Cons (x1,v) e -> x2 -> F -> v') where  -- (S'3)
         select' (Cons (x1,v) e) x2 F = select e x2

rule Select' (Cons (a, b) e -> c -> T -> d) ==> b=d

-- things we couldn't do with FDs
-- stronger improvement rules

rule Select (Nil->l->a) ==> False
rule Select' (Nil->l->t->a) ==> False
rule Select' ((Cons (x1,v1) e)->x2->T->v2) ==> x1=x2, v1=v2

------ record extension ------

-- we don't check whether a field already exists
-- usage: extend (Label,Entry) Record --> Record


-- class declaration

rule Extend a ==> a=(l,x)->r->r'

-- FD
rule Extend (a->b->r), Extend(a->b->r') ==> r=r'

-- instances
instance Extend ((l,x)->xs->Cons (l,x) xs) where
         extend (l,x) xs = Cons (l,x) xs

-- instance improvement
rule Extend ((l,x)->xs->ys) ==> ys=Cons (l,x) xs
```

```
------ check that all record labels are unique ------

-- usage: check Record --> Record
-- the identify operation if record is unique, otherwise type error

instance (Unique x) => Check (x->x) where
         check x = x

rule Check a ==> a=t->t

instance Unique Nil where
         unique = undefined

instance (Unique ys, Unique' (l,ys)) => Unique (Cons (l,x) ys) where
         unique = undefined

instance Unique' (l,Nil) where
         unique' = undefined

instance (EqR (l->l'->a), Fail a, Unique' (l',ys)) => Unique' (l', Cons (l,x) ys) where
         unique' = undefined

instance Fail F where
         fail = undefined

rule Fail T ==> False

------ field removal ------

-- we don't check whether a field doesn't exist
-- usage: remove Label Record --> Record

-- class declaration
rule Remove a ==> a=l->r->r'

-- FD
rule Remove (l->r->r'), Remove (l->r->r'') ==> r'=r''

-- instances
instance Remove (l -> Nil -> Nil) where
         remove l Nil = Nil

instance (EqR (l->l'->a), Remove' (a->l->(Cons (l',v) r)->r'))
                 => Remove (l -> (Cons (l',v) r) -> r') where
         remove l (Cons (l',v) r) = remove' (eqR l l') l (Cons (l',v) r)

-- instance improvements
```

```
rule Remove (l->Nil->r') ==> r'=Nil

-- class declaration
rule Remove' x ==> x=t->l->r->r'

-- FD
rule Remove' (t->l->r->r1), Remove' (t->l->r->r2) ==> r1=r2

-- instances
instance Remove' (T -> l -> (Cons (l',v) r) -> r) where
         remove' T l (Cons (l',v) r) = r

instance Remove (l->r->r') => Remove' (F -> l -> (Cons (l',v) r) -> (Cons (l',v) r')) where
         remove' F l (Cons (l',v) r) = Cons (l',v) (remove l r)

-- instance improvement
rule Remove' (T->a->(Cons (b,c) d)->e) ==> e=d
rule Remove' (F->a->(Cons (b,c) d)->e) ==> e=Cons (b,c) e'

------ examples ------

l0 = Zero
l1 = Succ l0
l2 = Succ l1

data MyString = A | B | C
data Phone  = P1 | P2


r1 = Nil
--r2 :: Cons (Zero,MyString) Nil
r2 = extend (l0,A) r1
r2' = check r2

r3 = extend (l1,P1) r2
r3' = check r3
-- r3 = extend (l0,P1) r2 yields type error,
--  check r3 verifies that all labels are unique!

x :: MyString
x = select r2 l0

y = remove l0 r3

z = extend (l1, P2) r3
-- note that (check z) yields an error

-- to run this program
-- call "chameleon --no-term --no-rr -o record.hs record.ch"
```

```
--  --no-term switches off the termination checker
--  --no-rr allows for non-range-restricted CHRs
```

6 Using Haskell from Chameleon

Programs written in Chameleon may make use of Haskell libraries directly, provided that an appropriate Chameleon interface file exists for each library. Library interfaces must be imported into a Chameleon program before the corresponding Haskell library can be used.

The keyword `interface Module`, allows you to access all of the functions and constructors defined in the Haskell module with the same name, provided that a corresponding interface file exists in the current directory. The interface file for Haskell module `Module` must be called `Module.ch`, and must be present in the current directory.

The following Chameleon program makes use of functions provided by the Haskell 98 Prelude.

```
interface Prelude
myReverse = foldl (flip (:)) [] <br>
palin xs = myReverse xs == xs <br>
```

In our translation of Chameleon programs, we simply replace `interface` by `import` statements. The Haskell 98 Standard Prelude interface file

`http://www.comp.nus.edu.sg/~sulzmann/chameleon/interface/Prelude.ch`

is distributed along with the Chameleon source code. Interfaces for other libraries remain to be created.

6.1 Interface Files

Interface files are nothing more than Chameleon source files. From the point-of-view of the compiler, an `interface` declaration causes the contents of the interface file to be treated as though they were part of the one program. Consequently, interface files should declare only the types of functions or constructors they provide, since their implementation is already part of an existing Haskell library. Note that we assume that inferface files must be in the same directory as the Chameleon program.

The `extern` keyword can be used to declare the types of both functions and data constructors without providing a definition. It may also be used to declare type constructors.

Its two forms are:

- `extern` *sig*: Declares a function/constructor, with the given type, e.g. `extern fst :: (a,b) -> a`, declares a function `fst` with the usual type; `extern Just :: a -> Maybe a` , declares the `Just` constructor of the standard Haskell `Maybe` type.
- `extern` *id*: Declares a new type constructor, *id*, e.g. `extern Maybe` declares the `Maybe` type constructor which could then be used to declare the type of `Just` as above.

Note that the Chameleon system currently does not perform any kind checking.

Consider the following Haskell library code (taken from the Haskell 98 Prelude Documentation):

```
data  Maybe a  =  Nothing | Just a
maybe              :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  =  n
maybe n f (Just x) =  f x
```

To make use of this, Chameleon requires an interface file containing the following:

```
extern Maybe
extern Nothing :: Maybe a
extern Just :: a -> Maybe a
extern maybe :: b -> (a -> b) -> Maybe a -> b
```

Note that the correctness of the interface with respect to the original Haskell code must be ensured by the user.

Haskell class and instance declarations must be manually translated to CHRs. For details we refer to [SS02]. We plan to incorporate an automatic translation in the near future. All Haskell constraints (type class constraints) must also be declared using `hconstraint TC` declarations. E.g. the interface to a library declaring a type class `Eq`, must contain the declaration: `hconstraint Eq`. Note that the user must guarantee that the number of parameters for each type class are consistent throughout the program.

6.2 Basic Types and Literals in Chameleon

Chameleon has built-in support for:

- Character literals of form: *'char'*
- String literals, like `"A string."`
- Numeric literals, both integer and floating point.
- Boolean values, `True` and `False`

Any of these may appear as terms or as patterns.

The following types are also available by default: function (infix) `t1->t2`, list `[t]`, and tuples up to 7 arguments ; and basic types `Bool`, `Int`, `Char`. Note that the Prelude interface provides access to all of the standard Haskell types - although their corresponding values might not be easily expressed in Chameleon for now (e.g. Rationals.)

The following Haskell type class constraints are also known to Chameleon: `Num`, `Fractional`, `Enum` and `Monad`. No instances are declared by default - the standard Prelude interface provides all the type class instances you'd expect in Haskell 98.

6.3 Missing Haskell Features

Although Chameleon is closely related to Haskell, not all Haskell language features are supported in Chameleon. Notable omissions from Chameleon include:

- `class` and `instance` declarations
- the module system (Chameleon has only the simple notion of "interfaces" as mentioned above)
- record syntax for data types

Please note also the differences between Haskell and Chameleon type inference as outlined earlier.

Support for these (and other features) may be added in future.

6.4 Mixing Haskell and Chameleon Overloading

Though, we cannot define full Haskell classes and instances we can make use of Chameleon's overloading mechanism. Indeed, as long as all Haskell constraints are declared as such `hconstraint`, we can freely mix the two forms of overloading.

Consider the following Chameleon program.

```
interface Prelude
overload insert
rule Insert x ==> x=e->ce->ce
instance Ord a => Insert (a->[a]->[a]) where
   insert x [] = [x]
   insert x (y:ys) = if x < y then x:y:ys else y:(insert x ys)
```

Note that `Ord` is a Haskell type class whereas `Insert` refers to a overloaded Chameleon identifier. When translating such mixed code, we perform evidence translation of overloaded Chameleon identifiers. However, Haskell type classes are left untouched and will be dealt with when running the translated Chameleon program through a Haskell compiler.

In fact, by mixing Haskell and Chameleon overloading via interfaces allows us to directly express the zip and record Example in terms of a user-definable Haskell extension! Assume in `MZip.hs` we find the following Haskell code.

```
-------------
-- MZip.hs --
-------------
module MZip where
  mzip2 :: [a]->[b]->[(a,b)]
  mzip2 (a:as) (b:bs) = (a,b) : (mzip2 as bs)
  mzip2 _      _      = []

  class MZip a b c | a c -> b, b c -> a where
      mzip :: [a] -> [b] -> c
  instance MZip a b [(a,b)] where
      mzip = mzip2
  instance MZip (a,b) c e => MZip a b ([c]->e) where
      mzip as bs cs = mzip (mzip2 as bs) cs
```

Assume in `MZip.ch` we find the following interface specification which correctly describes the class and instance relations from above.

```
-------------
-- MZip.ch --
-------------
hconstraint MZip
extern mzip :: MZip a b c => [a] -> [b] -> c
rule MZip a b c, MZip a d c ==> b=d       -- (FD1)
rule MZip a b c, MZip d b c ==> a=d       -- (FD2)

rule MZip a b [(a,b)] <==> True             -- (Inst1)
rule MZip a b ([c]->e) <==> MZip (a,b) c e  -- (Inst2)

-- instance improvement linked to (Inst1)
```

```
rule MZip a d [(a,b)]      ==> d=b         -- (Impr1a)
rule MZip d b [(a,b)]      ==> d=a         -- (Impr1b)

-- note that instance improvement for (Inst2) is trivial
```

Then, we can define our own Haskell extension as follows.

```
--------------------
-- MZip-strong.ch --
--------------------
interface Prelude
interface MZip

-- we're strengthening rules (Impr1a) and (Impr1b)
rule MZip a d [c] ==> c=(a,b), d=b
rule MZip d b [c] ==> c=(a,b), d=a

e2 = head (mzip [True,False] ['a','b','c'])
e3 = head (mzip [True,False] ['a','b','c'] [1::Int,2])
```

Chameleon translates the above program as follows (via `chameleon -o MZip-strong.hs MZip-strong.ch`).

```
import Prelude
import MZip

e2 :: (Bool,Char)
e2 = head (mzip [True,False] ['a','b','c'])
e3 :: ((Bool,Char),Int)
e3 = head (mzip [True,False] ['a','b','c'] [1::Int,2])
```

That is, in our own Haskell extension we could sufficiently improve types such that a Haskell compiler can finally run the above program.

7 Running Chameleon Programs

Chameleon programs are translated into Haskell 98, which must be further compiled, before they can be run. The process is as follows.

- Run `chameleon` *`InputFile.ch`* `-o` *`OutputFile.hs`*
- Run your favorite Haskell 98 interpreter/compiler on the resulting output file.
  e.g. `hugs` *`Outfile.hs`*

For a brief description of available command line options, run `chameleon --help`.

References

[Frü95] T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.

[GHC] Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

[GJDS04] P. J. Stuckey G. J. Duck, S. Peyton-Jones and M. Sulzmann. Sound and decidable type inference for functional dependencies. In *Proc. of ESOP'04*, volume 2986 of *LNCS*, pages 49–63. Springer-Verlag, 2004.

[Has] Haskell 98 language report. http://research.microsoft.com/Users/simonpj/haskell98-revised/haskell98-report-html/.

[HUG] Hugs home page. haskell.cs.yale.edu/hugs/.

[JN00] M. P. Jones and J. Nordlander, 2000. http://haskell.org/pipermail/haskell-cafe/2000-December/001379.html.

[Jon00] M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

[SS02] P. J. Stuckey and M. Sulzmann. A theory of overloading. In *Proc. of ICFP'02*, pages 167–178. ACM Press, 2002.

[SSW03a] P. J. Stuckey, M. Sulzmann, and J. Wazny. The Chameleon type debugger. In *Proc. of Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, pages 247–258. Computer Research Repository (http://www.acm.org/corr/), 2003.

[SSW03b] P.J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in Haskell. In *Proc. of Haskell Workshop'03*, pages 72–83. ACM Press, 2003.

[SSW04] P. J. Stuckey, M. Sulzmann, and J. Wazny. Type annotations in Haskell, 2004. submitted for publication.

[SW] M. Sulzmann and J. Wazny. Implementing overloading in Chameleon. `http://www.comp.nus.edu.sg/~sulzmann/chr/download/imp.ps.gz`.