A Compiler for Constraint Handling Rules based on Partial Evaluation

Christian Holzbaur
University of Vienna
Department of Medical Cybernetics and Artificial Intelligence
Freyung 6, A-1010 Vienna, Austria
christian@ai.univie.ac.at

Thom Frühwirth
CWG at LMU*
Oettingenstrasse 67, D-80538 Munich, Germany
fruehwir@informatik.uni-muenchen.de

July 30, 1999

Abstract

We argue to base the compilation of a particular language for writing constraint solvers, CHR, on partial evaluation (PE). In contrast to previous compilers realizations the semantics of the generated code are easier to prove to correspond to the intended ones and we can capitalize on the possibility to control the degree of PE, which allows for everything between interpreted execution to full compilation. The resulting differences in execution speed and code size are subject to a tradeoff made and controlled by the user through compiler opotions. Further, PE based compilation prepares a next step in the evolution of CHR implementations where the realization of parts of the constraint store will be made available to the PE process. The specialization of the abstract data types that make up the constraint store is instrumental in narrowing the performance gap between special purpose constraint solvers and CHR.

1 Introduction

In this paper we focus on the compilation of a declarative language, especially designed for writing constraint solvers, called constraint handling rules (CHR) [Fru91, FrBr95, Fru98, FAM98, HoFr98]. With CHR, one can introduce user-defined constraints into a given high-level host language, be it Prolog or Lisp. As language extension, CHR themselves are only concerned with constraints, all auxiliary computations are performed in the host language.

CHR is essentially a committed-choice language consisting of guarded rules that rewrite constraints into simpler ones until they are solved. CHR can define both simplification of and propagation over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence. Propagation adds new constraints which are logically redundant but may cause further simplification.

*Constraint Working Group at Ludwig-Maximilians-University

In contrast to the family of the general-purpose concurrent logic programming languages [Sha89], concurrent constraint languages [Sar93] and the ALPS [Mah87] framework, CHR are a special-purpose language concerned with defining declarative objects, constraints, not procedures in their generality. In another sense, CHR are more general, since they allow for *multiple heads*, i.e. conjunctions of constraints in the head of a rule. Multiple heads are a feature that is essential in solving conjunctions of constraints. With single-headed CHR alone, unsatisfiability of constraints could not always be detected (e.g `X<Y,Y<X`) and global constraint satisfaction could not be achieved.

CHR are typically realized as a library containing a compiler, runtime system and solvers written in CHR. Here we are concerned exclusively with the compilation process. The design of the runtime system is covered in [HoFr98b, HoFr98c].

The main threads of motivation behind our proposal to base the transformation of CHR into an executable (high-level) language, Prolog in this case, on partial evaluation (PE) are:

- Our experience with earlier versions of the system indicated that the transformation is pretty expansive in terms of code size, even when targeting at a high-level language. The phenomenon does not hamper the utilization of constraint solvers consisting of a couple of CHR, the ratio between lines of code in the target vs. source language being roughly 10. But then, CHR directly and fairly efficiently implement production rule systems, often having hundreds of rules or more.
- We want a firm formal basis for the compiler.
- Once we open the black box that implements the constraint store upon which CHR act, the specialization of the corresponding abstract data types via PE will be instrumental in narrowing the residual performance gap between special purpose constraint solvers and CHR.

Overview of this Paper

We quickly recapitulate syntax and semantics for CHR. We implement the semantics in a Prolog context through the provision of an interpreter and derive the compiled code via PE. An example will guide us through the paper. Even though it does not define a typical constraint, we chose it for didactic reasons.

Example 1.1 (Primes) *We implement the sieve of Eratosthenes to compute primes in a way reminiscent of the "chemical abstract machine" [BCL88]: The constraint* `candidates(N)` *generates candidates for prime numbers,* `prime(M)`*, where* `M` *is between* `1` *and* `N`*. The candidates react with each other such that each number absorbs multiples of itself. In the end, only prime numbers remain.*

```
candidates(1) <=> true.                                    % rule 1
generate @ candidates(N) <=> N>1 |                         % rule 2
             M is N-1, prime(N), candidates(M).

sieve @ prime(I) \ prime(J) <=> J mod I =:= 0 | true.      % rule 3
```

The first rule says that the number `1` *is not a good candidate for a prime,* `candidates(1)` *is thus rewritten into* `true`*, a constraint that is always satisfied and therefore it has no effect and representation in the constraint store. Note that head matching is used in CHR so the first rule will only apply to* `candidates(1)`*. A constraint for* `candidates` *with a free variable, like* `candidates(X)`*, will suspend (delay).*

The `generate` *rule generates a candidate* `prime(N)` *and proceeds recursively with the next smaller number, provided the guard (precondition, test)* `N>1` *is satisfied.*

The third, multi-headed rule named `sieve` *reads as follows: If there is a constraint* `prime(I)` *and some other constraint* `prime(J)` *such that* `J mod I =:= 0` *holds, i.e.* `J` *is a multiple of* `I`*, then keep* `prime(I)` *but remove* `prime(J)` *and execute the body of the rule,* `true`*.*

2 Syntax and Semantics

We assume some familiarity with (concurrent) constraint (logic) programming, e.g. [Sha89, vHSD92, Sar93, JaMa94, MaSt98]. As a special purpose language, CHR extend a host language with (more) constraint solving capabilities. Auxiliary computations in CHR programs are executed as host language statements. Here the host language is (SICStus) Prolog. For more formal and detailed syntax and semantics of constraint handling rules see [Fru98, FAM98].

2.1 Syntax

Definition 2.1 *There are three kinds of CHR. A* simplification CHR *is of the form*

```
[Name '@'] Head1,...,HeadN '<=>' [Guard '|'] Body.
```

where the rule has an optional `Name`*, which is a Prolog term, and the multi-head* `Head1,...,HeadN` *is a conjunction of CHR constraints, which are Prolog atoms. The guard is optional; if present,* `Guard` *is a Prolog goal excluding CHR constraints; if not present, it has the same meaning as the guard* `'true |'`*. The body* `Body` *is a Prolog goal including CHR constraints*

A propagation CHR *is of the form*

```
[Name '@'] Head1,...,HeadN '==>' [Guard '|'] Body.
```

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

```
[Name '@'] Head1,...'\'...,HeadN '<=>' [Guard '|'] Body.
```

where the symbol `'\'` *separates the head constraints into two nonempty parts.*

A simpagation rule combines simplification and propagation in one rule. The rule `HeadsK \ HeadsR <=> Body` is equivalent to the simplification rule `HeadsK, HeadsR <=> HeadsK, Body`, i.e. `HeadsK` is kept while `HeadsR` is removed. However, the simpagation rule is more compact to write, more efficient to execute and has better termination behaviour than the corresponding simplification rule.

2.2 Semantics

Declaratively[1], a rule relates heads and body provided the guard is true. A simplification rule means that the heads are true if and only if the body is satisfied. A propagation rule means that the body is true if the heads are true.

In this paper, we are interested in the operational semantics of CHR in actual implementations. A CHR constraint is implemented as both *code* (a Prolog predicate) and *data* (a Prolog term) in the constraint store, which is a data structure holding constraints. Every time a CHR constraint is posted (executed) or woken

[1] Unlike general committed-choice programs, CHR programs can be given a declarative semantics since they are only concerned with defining constraints, not procedures in their generality.

(reconsidered), it checks itself the applicability of the rules it appears in. Such a constraint is called *(currently) active*, while the other constraints in the constraint store that are not executed at the moment are called *(currently) passive*.

Heads. For each CHR, one of its heads is matched against the constraint. Matching succeeds if the constraint is an instance of the head, i.e. the head serves as a pattern. If a CHR has more than one head, the constraint store is searched for *partner* constraints that match the other heads. If the matching succeeds, the guard is executed. Otherwise the next rule is tried.

Guard. A guard is a precondition on the applicability of a rule. The guard either succeeds or fails. A guard succeeds if the execution succeeds without causing an instantiation error[2] and without *touching* a variable from the heads. A variable is *touched* if it takes part in a unification or gets more constrained by a built-in constraint. If the guard succeeds, the rule applies. Otherwise it fails and the next rule is tried.

Body. If the firing CHR is a simplification rule, the matched constraints are removed from the store and the body of the CHR is executed. Similarly for a firing simpagation rule, except that the constraints that matched the heads preceding `'\'` are kept. If the firing CHR is a propagation rule the body of the CHR is executed without removing any constraints. It is remembered that the propagation rule fired, so it will not fire again (and again) with the same constraints. Since the currently active constraint has not been removed, the next rule is tried.

Suspension. If all rules have been tried and the active constraint has not been removed, it suspends (delays) until a variable occurring in the constraint is touched. Here suspension means that the constraint is inserted into the constraint store as data.

3 Implementation

The transformation scheme outlined in this section maps CHR to Prolog clauses which will (re)produce the semantics from the previous section. The mapping itself is coded in (SICStus) Prolog ([HoFr98]) and translates CHR into Prolog on-the-fly, while the CHR file is loaded (consulted). In contrast to earlier presentations based on definite clause grammar (DCG) templates ([HoFr98b]) we describe the mapping as the result of the partial evaluation of a CHR interpreter with respect to a given set of rules. A compiler that generates code that provably mirrors the interpreters semantics can be synthesized via PE. Besides this theoretical attractive construction, the degree of PE can be controlled and provides the user with a means to choose between compact interpreted and fully specialized, fast, yet voluminous code.

3.1 Transformation scheme, Interpreter skeleton

CHR compilation is non-local, i.e. not rule by rule, in the sense that we have to account for each constraint in each of the two possible roles (active,passive) in all rules where it occurs as a head. Roughly, we thread the flow of control through the match-attempts, guard evaluation, updates to the constraint store depending on the rule type, and evaluation of the rule bodies.

3.1.1 Global flow

Initially, the relation `chr_crossref/2` between constraints and rules is computed. It associates each constraint with all its occurrences in the rules. The order follows

[2] A built-in predicate of Prolog complains about free variables where it needs instantiated ones.

the one in the source file with the only exception that occurrences where the current constraint would be removed if the rule applied are listed first. The motivation is that this constellation is particularly efficient to execute (section 3.1.2). Example: With a current constraint `prime/1` and rules as listed in (example 1.1) we start with the second head in rule 3, followed by the first head in rule 3.

```
chr_crossref( prime(_),      [3:2,3:1]).
chr_crossref( candidates(_), [1:1,2:1]).
```

Example 3.1 (Interpreter skeleton) *Here is the part of the CHR interpreter that implements the crude execution plan: For a given constraint* `C` *its occurrences are determined via* `chr_crossref/2` *and processed in sequence. The constraint is suspended once the agenda gets empty.*

```
chr( C) :-
        chr_crossref( C, Agenda),       % Agenda = occurrences of C
        chr( Agenda, C).

chr( [],               C) :- suspend( C).
chr( [Current|Agenda], C) :-
        rule_parts( Current, ActiveHead, Partners, Guard, Body),
        ( subsumes( ActiveHead, C),
            join( Partners, ...),       % find partner constraints
            check_guard( Guard, ...) -> % commit
            ...                         % update store,
            call( Body)
            ...                         % maybe proceed with Agenda
        ;
            chr( Agenda, C)             % try next
        ).
```

If we have the set of rules available at compile time, partial evaluation of this part of the interpreter is trivial. PE of `chr_crossref/2` in particular reduces to simple *execution* at compile time for a given constraint `C`. Unrolling the recursion driven by the first argument in `chr/2` yields the residual clauses[3] (check example 3.2). The predicate `rule_parts/5` retrieves components like the active and partner heads, the guard, and the body for a given rule plus active head indication. With this data, PE will specialize `subsumes/2`, `join`, and `check_guard`[4].

Example 3.2 (Primes, contd.) *For the constraint* `candidates/1` *we generate the following residual code (edited for readability).*

```
% for each occurrence of the constraint as a head of a rule:

% in rule candidates(1) <=> true
candidates(A) :-
        nonvar(A),                      % matching
        A=1,                            %   instructions
        !.                              % commit

% in rule candidates(N) <=> N>1 | M is N-1, prime(N), candidates(M)
candidates(A) :-
        ground(A),                      % guard
        A>1,                            %   evaluation
        !,                              % commit
        C is A-1,                       % body
```

[3] Redundant term structure is removed by PE. For example, `chr(candidates(A),...) :- ...` becomes `candidates(A) :- ...`

[4] Definitions omitted due to space limit

```
        prime(A),                       % body
        candidates(C).                  % body

% if no rule applied, suspend the constraint on its variables
candidates(A) :-
        suspend(candidates(A)).
```

3.1.2 Join computation, Matching

The realization of the condition part of CHR consists of finding sets of tuples in the constraint store which match the heads of a rule, resembling the relational algebra join operation. In the binding environments of these tuple sets, the applicability of a rule is decided by evaluating its guard. The situation is quite similar to the matching phase in rule/production systems, where the earlier predominant state-preserving RETE match algorithm [For82] was redeemed by the superior state-less TREAT algorithm [Mir87]. State preservation is even more of debatable utility in the presence of guards. Thus, the CHR compilation draws upon a state-less incremental matching mechanism.

In the terminological framework of [LiOk87] CHR operate under the "immediate update view": at the time a rule commits, the constraints to be removed as indicated by the rule type are gone. Constraints added by a rule's body are visible to the currently active rule(s) immediately. There are three prototypical cases [HoFr98c]:

1. The active constraint is removed by the rule. Independent from the number of partner constraints, the rule will apply at most once, and no further rules will have to be tried since the active constraint can no longer participate in a join.

2. The active constraint is kept, some partner constraints are removed. Since the active constraint is kept, one has to continue looking for applicable rules after the rule applied. However, since at least one partner constraint will have been removed, the same rule will only be applicable again with another constraint from the store that matches the same partner head.

3. The active constraint and all partner constraints are kept. This is the most expensive case. The full crossproduct of constraints as indicated by the rule heads has to be generated. Further rules will have to be tried since the active constraint is not removed.

Example 3.3 (Primes, contd.) *The residual code for* `prime/1` *from our example demonstrates cases 1,2 from above.* `nd_init_iteration/4` *nondeterministically generates candidate constraints with the given functor and arity (*`prime/1`*) from the constraint store:*

```
% rule prime(I) \ prime(J) <=> J mod I =:= 0 | true.

prime(J) :-                     % 2nd head in rule 3, case 1
        nd_init_iteration(prime, 1, _, I),
        ground(I),
        ground(J),
        J mod I =:= 0 ,         % J is a multiple of some prime in the store
        !.                      % commit, thus J not stored
prime(I) :-
        prime_loop(I),          % remove multiples of I
        suspend( prime(I)).     % store

prime_loop(I) :-                % 1st head in rule 3, case 2
        nd_init_iteration(prime, 1, D, J),
```

```
        ground(I),
        ground(J),
        J mod I =:= 0, !,    % J is a multiple of I
        remove_constraint(D),% remove prime(J)
        prime_loop(I).       % other multiples of I?
prime_loop(_).               % exhausted
```

3.2 Specialized Partial Evaluation

We started our PE experiments on CHR with MIXtus [Sah91], a partial evaluator for full Prolog. The formulation of the interpreter had to be revised a couple of times to yield the desired residual code. Although MIXtus should be self-applicable in principle and thus allow for Futamura projections [Futa71], i.e. the automated formal synthesis of a CHR compiler from the CHR interpreter, we manually derived a compiler from the CHR interpreter. The main reasons are:

- We need only a fraction of the MIXtus coverage of Prolog to translate CHR.
- PE for full Prolog probably does not meet the performance requirements of a production compiler - even after auto-projection.
- We would have had to teach MIXtus about the partial evaluation of subsumption, i.e. matching. Providing MIXtus with the corresponding predicates results in rather slow PE. Directly extending the MIXtus set of built-in predicates exceeds our expertise on the non-documented internals of MIXtus.
- We require quite detailed control over some aspects of the residual code for ultimate performance. As a particular example consider the minimization of argument motion: If a predicate calls another predicate, notably recursively, and some arguments are passed on, we want them at the same argument positions in caller and callee to avoid logically irrelevant but runtime consuming register copy instructions at the WAM level.

4 Conclusions

The overall effect of the PE approach is that we can now *derive* the compiler in a sound fashion. This is to be seen in contrast to earlier versions of the compiler which were based on ad hoc DCG templates for the three prototypical cases from section 3.1.2. Under the assumption that demonstration of correctness is sufficient and simpler on the more compact representation of the CHR interpreter, PE exonerates the compiler generation from that burden. It is also quite re-assuring to see that the earlier templates are indeed roughly re-synthesized through PE.

Among the plans for the future development of the CHR implementation is the specification of the constraint store as an abstract data type (ADT). The default implementation would be the current one based on suspensions via attributed variables. Through the provision of specialized implementations on a constraint by constraint basis, the user can exploit peculiarities of his/her application. If all the constraints of one type are ground for example, they make no reference to the suspend/wake mechanism. In that case this part of the store is better kept in a relational data base, which quite likely provides indices to facilitate the join computations.

References

[BCL88] Banatre J.-P., Coutant A. and Le Metayer D., A Parallel Machine for Multiset Transformation and its Programming Style, Future Generation Computer Systems 4:133-144, 1988.

[CaWi95] Carlsson M., Widen J, Sicstus Prolog Users Manual, Release 3#0, Swedish Institute of Computer Science, SICS/R-88/88007C, 1995.

[For82] Forgy C.L, Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, Artificial Intelligence, 19(1), 17-37, 1982.

[FrBr95] T. Frühwirth and P. Brisset, High-Level Implementations of Constraint Handling Rules, Technical Report ECRC-95-20, ECRC Munich, Germany, June 1995.

[FAM98] T. Frühwirth, S. Abdennadher and H. Meuss, Confluence and Semantics of Constraint Simplification Rules, Constraint Journal, Kluwer Academic Publishers, 1998.

[Fru91] T. Frühwirth, Introducing Simplification Rules, Technical Report ECRC-LP-63, ECRC Munich, Germany, October 1991.

[Fru98] T. Frühwirth, Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming (P. Stuckey and K. Marriot, Eds.), Journal of Logic Programming, Vol 37(1-3), pp 95-138, October 1998.

[Futa71] Y. Futamura, Partial Evaluation of Computation Process – An approach to a Compiler-Compiler, in: Systems, Computers, Controls, 2(5)45-50, 1971.

[HoFr98] Ch. Holzbaur C. and Th. Frühwirth, Constraint Handling Rules Reference Manual, for SICStus Prolog, Österreichisches Forschungsinstitut für Artificial Intelligence, Vienna, Austria, TR-98-01, March 1998.

[HoFr98b] Ch. Holzbaur and Th. Frühwirth, Compiling Constraint Handling Rules, ERCIM/COMPULOG Workshop on Constraints, CWI, Amsterdam, The Netherlands, 1998.

[HoFr98c] Ch. Holzbaur C. and Th. Frühwirth, Join Evaluation Schemata for Constraint Handling Rules, 13th Workshop Logische Programmierung WLP'98, TU Vienna, Austria, September 1998.

[JaMa94] J. Jaffar and M. J. Maher, Constraint Logic Programming: A Survey, Journal of Logic Programming, 1994:19,20:503-581.

[LiOk87] Lindholm T. O'Keefe R.A.: Efficient Implementation of a Defensible Semantics for Dynamic PROLOG Code, in Lassez J.L.(ed.), Proceedings of the Fourth International Conference on Logic Programming - Volume 1, MIT Press, Cambridge, MA, pp.21-39, 1987.

[Mah87] Maher M. J., Logic Semantics for a Class of Committed-Choice Programs, Fourth Intl Conf on Logic Programming, Melbourne, Australia, MIT Press, pp 858-876.

[MaSt98] K. Marriott and P. J. Stuckey, Programming with Constraints, MIT Press, USA, March 1998.

[Mir87] Miranker D.P.: TREAT: A Better Match Algorithm for AI Production Systems, in Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI- 87), Morgan Kaufmann, Los Altos, CA, pp.42-47, 1987.

[Sah91] Sahlin D, An Automatic Partial Evaluator for Full Prolog, Swedish Institute of Computer Science, 1991.

[Sar93] V. A. Saraswat, Concurrent Constraint Programming, MIT Press, Cambridge, 1993.

[Sha89] E. Shapiro, The Family of Concurrent Logic Programming Languages, ACM Computing Surveys, 21(3):413-510, September 1989.

[vHSD92] P. van Hentenryck, H. Simonis and M. Dincbas, Constraint Satisfaction Using Constraint Logic Programming, Artificial Intelligence, 58(1-3):113–159, December 1992.