Compiling Constraint Handling Rules into Prolog with Attributed Variables

Christian Holzbaur[1]* and Thom Frühwirth[2]

[1] University of Vienna
Department of Medical Cybernetics and Artificial Intelligence
Freyung 6, A-1010 Vienna, Austria
christian@ai.univie.ac.at
[2] Ludwig-Maximilians-University
Department of Computer Science
Oettingenstrasse 67, D-80538 Munich, Germany
fruehwir@informatik.uni-muenchen.de

Abstract. We introduce the most recent and advanced implementation of constraint handling rules (CHR) in a logic programming language, which improves both on previous implementations (in terms of completeness, flexibility and efficiency) and on the principles that should guide such a Prolog implementation consisting of a runtime system and a compiler. The runtime system utilizes attributed variables for the realization of the constraint store with efficient retrieval and update mechanisms. Rules describing the interactions between constraints are compiled into Prolog clauses by a multi-phase compiler, the core of which comprises a small number of compact code generating templates in the form of definite clause grammar rules.
Keywords: Logic and constraint programming, Implementation and compilation methods.

1 Introduction

In the beginning of constraint logic programming (CLP), constraint solving was "hard-wired" in a built-in constraint solver written in a low-level language. While efficient, this so-called "black-box" approach makes it hard to modify a solver or build a solver over a new domain, let alone debug, reason about and analyze it. This is a problem, since one lesson learned from practical applications is that constraints are often heterogeneous and application-specific. Consequently, several proposals have been made to allow more for flexibility and customization of constraint systems ("glass-box" or even "no-box" approaches):

- Demons, forward rules and conditionals in CHIP [6] allow the definition of propagation of constraints in a limited way.

* Part of this work was performed while visiting CWG at LMU with financial support from DFG.

- Constraint combinators in cc(FD) [13] allow to build more complex constraints from simpler constraints.
- Constraints connected to a Boolean variable in BNR-Prolog [2] and "nested constraints" [31] allow to express any logical formula over primitive constraints.
- Indexicals in clp(FD) [5] allow to implement constraints over finite domains at a medium level of abstraction.
- Meta- and attributed variables [26], [21], [15] allow to attach constraints to variables at a low level of abstraction.

It should be noted that all the approaches but the last can only extend a solver over a given, specific constraint domain, typically finite domains. The expressive power to realize other (application-specific) constraint domains is only provided by the last approach.

Attributed variables provide direct access storage locations for properties associated with variables. When such variables are unified, their attributes have to be manipulated. Thus attributed variables make unification user-definable [15], [16], [17]. Attributed variables require roughly the same implementation effort as hard-wired delay (suspension) and coroutining mechanisms found in earlier Prolog implementations, while being more general. And indeed, attributed variables nowadays serve as the primary low-level construct for implementing suspension (delay) mechanisms and constraint solver extensions in many constraint logic programming languages, e.g. SICStus [4] and ECL^{i}PS^e [3] Prolog. However writing constraints this way is tedious, a kind of "constraint assembler" programming.

If there already is a powerful constraint assembler, one may wonder what an associated high-level language could look like. Our proposal is a declarative language extension especially designed for writing constraint solvers, called constraint handling rules (CHR) [10], [12], [18], [11]. 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 have been used in dozens of projects worldwide to encode dozens of constraint handlers (solvers), including new domains such as terminological and temporal reasoning. If comparable hard-wired constraint solvers are available, the price to pay for the flexibility of CHR is often within an order of magnitude in runtime. The performance gap can in many cases be eliminated by tailoring the CHR constraints to the specifics of the class of applications at hand.

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. CHR can be seen as a generalization of the various CHIP [6] constructs for user-defined constraints.

In contrast to the family of the general-purpose concurrent logic programming languages [29], concurrent constraint languages [28] and the ALPS [23] 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. The probably most distinguishing functionality of CHR is that they act as a powerful iteration, retrieval, and upadte mechanism over the constraint store, a data structure holding constraints.

The first implementation of CHR in 1991 was an interpreter written in ECL^{i}PS^e Prolog. Then, the CHR language has been implemented in 1993 in Common LISP at the German Research Institute for Artificial Intelligence [14] and in 1994 as a library of ECL^{i}PS^e [9], [10]. A CHR interpreter was written in the concurrent logical object-oriented constraint language OZ [32] in 1996. Independent of our work, a new experimental prototype of CHR has been implemented recently in ECL^{i}PS^e 4.0 [30].

CHR are typically realized as a library containing a compiler, runtime system and solvers written in CHR. With Prolog as the host language, the idea is to realize the CHR constraint store through attributed variables. Rule application compiles into Prolog clauses which inspect and update the constraint store at runtime. Thus CHR can also be understood as a powerful means to manipulate the attributes of variables in a declarative high-level fashion. In this paper we introduce the most recent and advanced implementation of CHR in SICStus Prolog [18], which improves both on the previous implementation [10] in terms of completeness, flexibility and efficiency and on the principles that should guide such an implementation [9]. The new release also includes about 30 constraint solvers written in CHR.

For the user, the new release of CHR improves over older versions in the following aspects:

- The number of heads in a rule is no longer limited to two.
- Guards now with Ask and Tell as in concurrent constraint languages.
- Code runs generally about twice as fast as in older versions.
- For more control, rules are compiled in textual order.
- Compilation is now transparent to the user, on-the-fly when loading.
- Improved set of built-in predicates for advanced CHR users.
- Constant time access to constraints of one type for elevated performance.
- New options and pragmas for powerful compiler optimizations.
- Runtime system includes a stepper for Prolog-like debugging.

Similar issues, i.e. compilation of committed-choice languages into Prolog, have been investigated before, be it translating GHC [33], implementations of delay declarations [25] or the efficient implementation of QD-Janus [8]. Today, we benefit from more powerful programming constructs, in particular customizable suspension mechanisms provided by attributed variables. CHR specific topics are multiple heads and propagation rules.

Overview of this Paper We quickly recapture syntax and semantics for CHR. Then we describe the three phases of the new compilation scheme and the runtime system for CHR. We conclude with a comparison with the previous implementation. This paper is a revised version of [19].

An example will guide us through the paper. Even though it does not define a typical constraint, we chose it for didactic reasons. It is small but can still illustrate the various stages of our compilation scheme. We use Prolog syntax in this paper.

Example 1 (Primes). We implement the sieve of Eratosthenes to compute primes in a way reminiscent of the "chemical abstract machine" [1]: 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.
generate @ candidates(N) <=> N>1 | M is N-1, prime(N), candidates(M).

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

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. 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. [29], [13], [28], [22], [24]. 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 [12], [11].

2.1 Syntax

Definition 1. *There are three kinds of CHR. A* simplification CHR *is of the form*[1]

[1] For simplicity, we omit syntactic extensions like pragmas which are not relevant for this paper.

```
[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[2], 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 (reconsidered), it triggers checks to determine 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

[2] 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.

causing an instantiation error[3] 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 The Compiler

The compiler is written in (SICStus) Prolog [18] and translates CHR into Prolog on-the-fly, while the file is loaded (consulted). Its kernel consists of a definite clause grammar that generates the target instructions (clauses) driven by templates. We will use example 1 to explain the three phases of the compiler: (1) Parsing, (2) translating CHR into clauses using templates and (3) partial evaluation using macros. Of course, phase (2) is the essential one that encodes the algorithm.

3.1 Parsing Phase

Using the appropriate operator declarations, a CHR can be read and written as a Prolog term. Hence parsing basically reduces to computing information from the parse tree and to producing a canonical form of the rules. Information needed from the parse tree includes:

- The set of global variables, i.e. those that appear in the heads of a rule.
- The set of variables shared between the heads.

In the canonical form of the rules,

- each rule is associated with a unique identifier,
- rule heads are collected into two lists (named `Keep` and `Remove`), and
- guard and body are made explicit with defaults applied.

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

One list, called `Keep`, contains all head constraints that are kept when the rule is applied, the other list, called `Remove`, contains all head constraints that are removed. Lists may be empty. As a result of this representation, simplification, propagation and simpagation rules can be treated uniformly.

Example 2 (Primes, contd.). The canonical form of the rules for the prime number example is given below.

```
% rule(Id,Keep,      Remove,          Guard,         Body)

  rule( 1,[],        [candidates(1)], true,          true).
  rule( 2,[],        [candidates(A)], A>1,          (B is A-1,prime(A),
                                                     candidates(B))).
  rule( 3,[prime(A)],[prime(B)],     B mod A =:= 0, true).
```

3.2 Translation Phase

Each CHR constraint compiles into Prolog clauses that try the constraint with all rules in whose heads it occurs. The resulting compilation process is nonlocal in the sense that a CHR constraint may appear in various head positions in various rules. Each occurrence of a CHR constraint in the head of a rule gives rise to one clause for that constraint. The clause head contains the active constraint, while the clause body does the following:

- match formal parameters to actual arguments of head constraint
- find and match partner constraints
- check the guard
- commit via cut
- remove matched constraints if required
- execute body of rule

We first illustrate the compilation with a simple example, a single-headed simplification CHR, then we consider general cases of arbitrary multi-headed rules.

Example 3 (Primes, contd.). For the constraint `candidates/1` the compiler generates the following intermediate code (edited for readability).

```
% in rule candidates(1) <=> true
candidates(A) :-                        %  1
        match([1], [A]),                %  2
        check_guard([], true),          %  3
        !,                              %  4
        true.                           %  5

% in rule candidates(N) <=> N>1 | M is N-1, prime(N), candidates(M)
candidates(A) :-                        %  6
        match([C], [A]),                %  7
```

```
        check_guard([C], C>1),          %  8
        !,                              %  9
        D is C-1,                       % 10
        prime(C),                       % 11
        candidates(D).                  % 12

% if no rule applied, suspend the constraint on its variables
candidates(A) :-                        % 13
        suspend(candidates(A)).         % 14
```

The predicate `match(L1,L2)` matches the actual arguments `L2` against the formal parameters `L1`. The predicate `check_guard(VL,G)` checks the guard `G`. `check_guard/2` fails as soon as the global variables (list `VL`) are touched[4].

When no rule applied, the last clause inserts the constraint into the constraint store using a suspension mechanism. It allocates the suspension data structure and associates it with each variable occurring in the constraint. Touching any such variable will wake the constraint.

The real challenge left is to implement *multi-headed* CHR. In a naive implementation of a rule, the constraint store is queried for the cross-product of matching constraints. For each tuple in the cross-product the guard is checked in the corresponding environment. If the guard is satisfied, constraints that matched heads in the `Remove` list are removed from the store and the instance of the rule's body is executed. Note that the removal of constraints removes tuples from the cross-product.

Our implementation computes only those tuples in the cross-product that are really needed (as in [9]). Moreover, nondeterministic enumeration of the constraints is preferred over deterministic iteration whenever possible, because Prolog is good at backtracking [20].

For each head constraint in a rule the compiler does the following: It is deleted from the `Keep` or `Remove` list, respectively, and it is rendered as the *active* one. Whether the active constraint is removed when the rule applies, and whether any other head constraints are removed, leads to the following three prototypical cases, each covered by a code generating template in the compiler:

1. Case Active constraint from `Remove` list
2. Case Active constraint from `Keep` list, `Remove` list nonempty
3. Case Active constraint from `Keep` list, `Remove` list empty

Interestingly, the three cases do not directly correspond to the three kinds of CHR.

Case 1. Active constraint from `Remove` list The active head constraint is to be removed if the rule applies, so the rule under consideration is either a simplification or simpagation rule. It can be applied at most once with the

[4] In most Prolog implementations, it is more efficient to re-execute head matching and guards instead of suspending all of them and executing them incrementally.

current active constraint. The search for the partner constraints in this case can be done through nondeterministic enumeration. Here is the template as DCG grammar rule, slightly abridged. The predicate `ndmpc` generates the code to nondeterministically enumerate and match the partners, one by one.

```
compile(remove(Active), Remove, Keep, Guard, Body, ...) -->
        % compiler code
        {
          Active =.. [_|Args],
          same_length(Args, Actual),
          ...
          ndmpc(Remove, RemoveCode, RemCs, ...),
          ndmpc(Keep, KeepCode, ...)
        },
        % generated code
        [(constraint(head(F/A,R-N), args(Actual)) :-
                  match(Args, Actual),
                  RemoveCode,                % Identify Remove partners
                  KeepCode,                  % Identify Keep partners
                  check_guard(Vars, Guard),
                  !,
                  remove_constraints(RemCs),
                  Body
        )].
```

The variables `F,A,R` and `N` stand for functor, arity of the constraint, rule identifier and number of head in rule, respectively.

Example 4 (Primes, contd.). The second occurrence of prime/1 in rule 3 of Example 1 matches this template, and here is its instantiation:

```
% prime(I) \ prime(J) <=> J mod I =:= 0 | true.
constraint(head(prime/1,3-2), args([A])) :-
        match([C], [A]),
    % RemoveCode (for one partner constraint)
        get_constr_via([], Constraints),
        nd_init_iteration(Constraints, prime/1, Candidate),
        get_args(Candidate, [F]),
        match([C]-[G], [C]-[F]),
    % KeepCode (no partner constraints to be kept in this case)
        true,
    % Guard
        check_guard([G,C], (C mod G =:= 0)),
        !,
        remove_constraints([]),  % no constraints to remove here
    % Body
        true.
```

The predicate `get_constr_via(VL,Cs)` returns a handle `Cs` to the constraints suspended on a free variable occurring in the list `VL`. If there is no variable in `VL`, it returns a handle to all the constraints in the store. `nd_init_iteration(Cs, F/A, Candidate)` nondeterministically returns a candidate constraint with functor `F` and arity `A` through the handle `Cs`.

Case 2. Active constraint from `Keep` list, `Remove` list nonempty This case applies only if there is at least one constraint to be removed, but the active constraint will be kept. It can only originate from a simpagation rule. Since the active constraint is kept, one has to continue looking for applicable rules, even 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 in place of the removed one. Therefore, we can deterministically iterate over the constraints that are candidates for matching the corresponding head from `Remove`, while the remaining partners can be found via nondeterministic enumeration as before. At the end of the iteration, we have to continue with the remaining rules for the active constraint.

Example 5 (Primes, contd.). For space reasons, we just present a simple instance of the template, originating from the first occurrence of prime/1 in rule 3 (for readability with the predicate already flattened, as described in Section 3.3):

```
% rule prime(I) \ prime(J) <=> J mod I =:= 0 | true.
prime(A, B) :-
        get_constr_via([], C),          % get constraints from store
        init_iteration(C, prime/1, D),  % get partner candidates
        !,
        prime(D, B, A).                 % try to apply the rule

prime(A, B, C) :-
        iteration_last(A),              % no more partner candidate
        prime_1(C, B).                  % try next rule head
prime(A, B, C) :-
        iteration_next(A, D, E),        % try next partner candidate
        (  get_args(D, [F]),
           match([C]-[G], [C]-[F]),
           check_guard([C,G], (G mod C =:= 0))
        ->                              % rule applies
           remove_constraints([D]),     % remove the partner from store
        ;
           true                         % rule did not apply
        ),                              % in any case, try same rule
        prime(E, B, C).                 % with another partner candidate
prime_1(C, B) :- ...                    % code to try next rule head
```

Case 3. Active constraint from `Keep` list, `Remove` list empty This case originates from propagation rules. Since no constraint will be removed, all possible combinations of matching constraints have to be tried. The rule under

consideration may apply with each combination. Therefore, all the partners (not just one as in the previous case) have to be searched through nested deterministic iteration. No matter if and how often the rule was applicable, at the end we have to continue with the remaining rules for the active constraint as in Case 2.

Example 6. This propagation rule is part of an interval solver. `X::Min:Max` constrains `X` to be within lower and upper bounds `Min` and `Max`.

```
X le Y, X::MinX:MaxX, Y::MinY:MaxY ==> X::MinX:MaxY, Y::MinX:MaxY.
```

The propagation rule produces roughly the following code for `X le Y`.

```
X le Y :- le_1(X, Y).

le_1(X, Y) :-                          % active constraint (X le Y)
     get_constr_via([X], CXs),         % get constraints on X
     init_iteration(CXs, ::/2, PCXs), % get partner candidates
     !,
     le_1_0(PCXs, X, Y).               % try to apply the rule
le_1(X, Y) :-                          % rule was not applicable at all
     le_2(X, Y).                       % continue with next rule

le_2(X, Y) :-                          % no next rule
     suspend(X le Y).                  % done, suspend the constraint

le_1_0(PCXs, X, Y) :-                  % outer loop for X::MinX:MaxX
     iteration_last(PCXs),             % no more partner candidate
     le_2(X, Y).                       % continue with next rule
le_1_0(PCXs, X, Y) :-
     iteration_next(PCXs, CX, PCXs1), % try next partner candidate for X
     (   get_args(CX,...), match(...),% match arguments
         get_constr_via([Y], CYs),    % constraints on Y for next head
         init_iteration(CYs, ::/2, PCYs)
     ->
           le_1_1(PCYs, PCXs1, X, Y)  % try to apply the rule
     ;
           le_1_0(PCXs1, X, Y)        % try next partner candidate for X
     ).

le_1_1(PCYs, PCXs, X, Y) :-            % inner loop for Y::MinY:MaxY
     iteration_last(PCYs),             % no more partner candidate for Y
     le_1_0(PCXs, X, Y).               % continue with outer loop for X
le_1_1(PCYs, PCXs, X, Y) :-
     iteration_next(PCYs, CY, PCYs1), % try next partner candidate for Y
     (   get_args(CY,...), match(...),% match arguments
     ->                                % rule applies finally
           X::MinX:MaxY, Y::MinX:MaxY,% rule body
           le_1_1(PCYs1, PCXs, X, Y)  % continue, find another Y partner
     ;                                 % rule did not apply
           le_1_1(PCYs1, PCXs, X, Y)  % continue, find another Y partner
     ).
```

3.3 Partial Evaluation Phase

The translation granularity was chosen so that the generated code would roughly run as is, with little emphasis on efficiency coming from local optimizations and specializations. These are performed in the final, third phase of the compiler using a simple instance of partial evaluation (PE). It is performed by using macros as they are available in most Prolog systems, e.g. [4]. In contrast to approaches that address all aspects of a language in a partial evaluator such as Mixtus [27], our restricted form of PE can be realized with an efficiency that meets the requirements of a production compiler.

The functionalities of the main compiler macros:

- The generic predicates steering the iteration over partner constraints are specialized with respect to a particular representation of these multi sets.
- Recursions (typically iterations over lists) that are definite at compile time are unfolded at compile time.
- As in [33], head matching is specialized into unification instructions guarded by `nonvar/1` tests.
- The intermediate code uses redundant function symbols for the convenience of the compiler writers, e.g. to keep object, compiler and runtime-system variables visually apart. The redundant function symbols also help in type-checking the compiler. Redundant function symbols are absent in the target code. In particular, clause heads are flattened to facilitate clause indexing. For example, `constraint(head(prime/1,3-2), args([A]))` will be transformed into something like `prime1_3_2(A)`.

Example 7 (Primes, contd.). The macro expansion phase results in the following code for our example 3. The code for matching and guard checking has been inlined. The resulting trivial matchings (line 7), guards (line 3) and bodies (line 5) have been removed by PE.

```
% rule candidates(1) <=> true.
candidates(A) :-                    %  1
        A==1,                       %  2
        !.                          %  4
% rule candidates(N) <=> N>1 | M is N-1, prime(N), candidates(M).
candidates(A) :-                    %  6
        nonvar(A),                  %  8
        A>1,                        %  8
        !,                          %  9
        B is A-1,                   % 10
        prime(A),                   % 11
        candidates(B).              % 12
candidates(A) :-                    % 13
        suspend(candidates(A)).     % 14
```

4 The Runtime System

The code generated by the compiler utilizes Prolog since CHR compile into clauses. Thus e.g. memory management is already taken care of. There are however functionalities that are not provided directly by most Prolog implementations:

- We need means to suspend, wake and re-suspend constraint predicates.
- We need efficient access to suspended constraints in the store through different access paths.

The vanilla suspension mechanisms used by earlier CHR implementations addressed the first issue above, but did not optimize re-suspension. The second issue was partially ignored in that plain linear search in (parts of) the constraint store was used.

4.1 Suspensions

Typically, the attributes of variables are goals that suspend on that variable. They are re-executed (woken) each time one of their variables is touched. Via the attributed variables interface as found in SICStus or ECL^{i}PS^e Prolog the behaviour of attributed variables under unification is specified with a user-defined predicate. In the CHR implementation, suspended goals are our means to store constraints.

In more detail, the components of the CHR suspension data structure are:

- Constraint goal
- State of constraint
- Unique identifier
- Propagation history
- Re-use counter

The state indicates if the constraint is active or passive.[5] The unique identifier is used, together with the propagation history, to ensure termination for propagation rules. Each propagation rule fires at most once for each tuple formed by the set of matched head constraints. The re-use counter is incremented with every re-use of the suspension. It is used for profiling and some more subtle aspects of controlling rule termination outside the scope of this paper.

To optimize re-suspensions, we made the suspension itself an argument of the re-executed goal. Internally, each constraint has an additional argument. When first executed, the argument is a free variable. When the constraint suspends, this extra argument is bound to the suspension itself. When it runs again, the suspension mechanism now has a handle to the suspension and can update its state. Traces of this mechanism were removed from the listed code samples in this paper to avoid confusion.

[5] In actuality the granularity of states and transitions is more copious. The additional mechanics mainly address lazy constraint removal to anticipate the possibility of subsequent constraint re-introduction.

4.2 Access Paths

When a CHR searches for a partner constraint, a variable common to two heads of a rule considerably restricts the number of candidate constraints to be checked, because both partners must be suspended on this variable. Thus we usually access the constraint store by looking at only those constraints (cf. `get_constr_via/2`). We also know functor and arity of the partner. Consequently, we want direct access to the set of constraints of given functor/arity. Earlier implementations performed this selection by linear search over a part of the suspended constraints.

Access to data through a variable, and then functor/arity, is exactly the functionality provided efficiently by attributed variables. In our runtime system we map every functor/arity pair to a fixed attribute slot of a variable at compile time yielding *constant time* access to the constraints of one type. Only the arguments need to be matched at runtime.

5 Preliminary Empirics

Benchmarks are difficult, because the new implementation is in SICStus Prolog, while the previous one was in ECL^{i}PS^e Prolog. Attributed variables are implemented differently in these Prologs. That said, our inchoate measurements indicate that the new compiler produces code that is roughly twice as fast. Specifically, we compared our new SICStus 3#7 CHR implementation with the one in distribution with ECL^{i}PS^e 3.5.2, measuring the variation between the two Prolog implementations together with the actual CHR implementation differences. Times are given in seconds. ECL^{i}PS^e and SICStus were run on the same machine (a Sun workstation). In ECL^{i}PS^e, the solvers were compiled without debugger hooks[6]. We have two columns for SICStus: one for native code, one for emulated code. The last column relates *emulated* SICStus and ECL^{i}PS^e.

Benchmark	SICStus native	a) SICStus emulated	b) ECL^{i}PS^e	ratio a/b
solver bool				
deussen1 ulm027r1, all solutions	0.370	0.470	0.900	0.52
schur(10,_), all solutions	1.020	1.300	2.584	0.50
schur(13,_), 1st solution	0.230	0.290	1.233	0.24
schur(13,_), all solutions	2.040	2.520	7.483	0.34
bnqueens(8,L), 1st solution	1.240	1.500	9.817	0.15
testbl(5,L), all solutions	0.750	0.900	1.467	0.61
solver lists				
word problem, 1st solution	0.380	0.460	0.633	0.73
word problem, 2nd solution	2.940	3.660	4.717	0.78

The new CHR version was faster on all examples, the ratio new vs. old ranging from 0.15 to 0.78, averaging 0.5 with a standard deviation of 0.2. The boolean

[6] Option `nodbgcomp`.

constraint solver features several different kinds of constraints and consequently benefits more from the new data structures than the solver for lists (that basically allows for equality between concatenations of lists).

Most problems are well-known from the literature: The Deussen problem ulm027r1 was originally provided by Mark Wallace, Schur's lemma and Boolean n-queens by Daniel Diaz. The final one is a puzzle of unknown origin posted by Bart Demoen in the newsgroup comp.lang.prolog. The word problem was provided by Klaus Schulz.

6 Conclusions

With the CHR system outlined in this paper we aimed at improvements in terms of completeness, flexibility and efficiency.

With regard to completeness some former limitations were removed:

- The number of heads in a rule is no longer limited to two. The restriction was motivated originally by efficiency considerations since more heads need more search time. One can encode rules with more than two heads using additional auxiliary intermediate constraints. But then, the resulting rules are not only hard to understand, they are also less efficient than a true multi-headed implementation. In addition, rules apply now in textual order, which gives the programmer more control.
- Guards now support *Ask* and *Tell* [28]. In this way, CHR can also be used as a general-purpose concurrent constraint language. (In this paper we only considered *Ask* parts of guards.)
- Due to space limitations we also have not discussed *options* and *pragmas* in this paper - these are annotations to programs, rules or constraints that enable the compiler to perform powerful optimizations, that can sometimes make programs terminate or reduce their complexity class.

The gain in flexibility of the implementation proper can be attributed to the following facts:

- The CHR compiler has been "orthogonalized" by introducing three clearly defined compilation phases. Compilation is now on-the-fly, while loading. The template-based translation with subsequent macro-based partial evaluation allows for easy experimentation with different translation schemata. It created the elbow room for a rather quick implementation of various compiler options and pragmas. The system was implemented in four man-months. The compiler is 1100 lines of Prolog, the runtime system around 600, which together is less than half of the ECL^{i}PS^e implementation.
- CHR specific demands, such as access paths and suspension recycling, are taken care of explicitly through customized versions of the suspension mechanism.
- Attributed variables let us efficiently implement the generalized suspension mechanism needed for CHR at the *source level*. In particular, constant time

access to constraints of one type can now be provided, instead of the linear time access in previous implementations.

Plans for the future development of the CHR implementation are the introduction of a priority scheme, realized through a scheduler [33] that makes the order in which simultaneously applicable rules are executed explicit, and the factorization of common matching instructions [7].

More information about CHR is available at the CHR homepage
`http://www.informatik.uni-muenchen.de/~fruehwir/chr-intro.html`

References

1. Banatre J.-P., Coutant A., Le Metayer D., A Parallel Machine for Multiset Transformation and its Programming Style, Future Generation Computer Systems 4:133-144, 1988.
2. Benhamou F., Older W.J., Bell Northern Research, June 1992, Applying interval arithmetic to Integer and Boolean constraints, Technical Report.
3. Brisset P. et al., ECL^{i}PS^e 4.0 User Manual, IC-Parc at Imperial College, London, July 1998.
4. Carlsson M., Widen J., Sicstus Prolog Users Manual, Release 3#0, Swedish Institute of Computer Science, SICS/R-88/88007C, 1995.
5. Diaz D., Codognet P., A Minimal Extension of the WAM for clp(FD), in Warren D.S.(Ed.), Proceedings of the Tenth International Conference on Logic Programming, The MIT Press, Budapest, Hungary, pp.774-790, 1993.
6. Dincbas M. et al., The Constraint Logic Programming Language CHIP, Fifth Generation Computer Systems, Tokyo, Japan, December 1988.
7. Debray S., Kannan S., Paithane M., Weighted Decision Trees, in Apt K.R.(Ed.), Logic Programming - Proceedings of the Joint International Conference and Symposium on Logic Programming, MIT Press, Cambridge, MA, pp.654-668, 1992.
8. Debray S.K., QD-Janus : A Sequential Implementation of Janus in Prolog, Software—Practice and Experience, Volume 23, Number 12, December 1993, pp. 1337-1360.
9. Frühwirth T., Brisset P., High-Level Implementations of Constraint Handling Rules, Technical Report ECRC-95-20, ECRC Munich, Germany, June 1995.
10. Frühwirth T., and Brisset P., Chapter on Constraint Handling Rules, in ECL^{i}PS^e 3.5.1 Extensions User Manual, ECRC Munich, Germany, December 1995.
11. Frühwirth T., Abdennadher S., Meuss H., Confluence and Semantics of Constraint Simplification Rules, Constraint Journal, Kluwer Academic Publishers, to appear.
12. Frühwirth T., 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.
13. Hentenryck P.van, Simonis H., Dincbas M., Constraint Satisfaction Using Constraint Logic Programming, Artificial Intelligence, 58(1-3):113–159, December 1992.
14. Herbig B., Eine homogene Implementierungsebene für einen hybriden Wissensrepräsentationsformalismus, Master Thesis, in German, University of Kaiserslautern, Germany, April 1993.

15. Holzbaur C., Specification of Constraint Based Inference Mechanisms through Extended Unification, Department of Medical Cybernetics and Artificial Intelligence, University of Vienna, Dissertation, 1990.
16. Holzbaur C., Metastructures vs. Attributed Variables in the Context of Extensible Unification, In 1992 International Symposium on Programming Language Implementation and Logic Programming, pages 260–268. LNCS631, Springer Verlag, August 1992.
17. Holzbaur C., Extensible Unification as Basis for the Implementation of CLP Languages, in Baader F., et al., *Proceedings of the Sixth International Workshop on Unification*, Boston University, MA, TR-93-004, pp.56-60, 1993.
18. Holzbaur C., Frühwirth T., Constraint Handling Rules Reference Manual, for SICStus Prolog, Österreichisches Forschungsinstitut für Artificial Intelligence, Vienna, Austria, TR-98-01, March 1998.
19. Holzbaur C., Frühwirth T., Compiling Constraint Handling Rules (CHR), Third ERCIM/Compulog Network Workshop on Constraints, CWI Amsterdam, The Netherlands, September 1998.
20. Holzbaur C., Frühwirth T., Join Evaluation Schemata for Constraint Handling Rules, 13th Workshop Logische Programmierung WLP'98, TU Vienna, Austria, September 1998.
21. Huitouze S.le, A new data structure for implementing extensions to Prolog, in Deransart P. and Maluszunski J.(Eds.), Programming Language Implementation and Logic Programming, Springer, Heidelberg, 136-150, 1990.
22. Jaffar J., Maher M.J., Constraint Logic Programming: A Survey, Journal of Logic Programming, 1994:19,20:503-581.
23. 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.
24. Marriott K., Stuckey J.P, Programming with Constraints, MIT Press, USA, March 1998.
25. Naish L., Prolog control rules, Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, California, September 1985, pp. 720-722.
26. Neumerkel U., Extensible unification by metastructures, In Proc. of Metaprogramming in Logic (META'90), Leuven, Belgium, 1990.
27. Sahlin D., An Automatic Partial Evaluator for Full Prolog, Swedish Institute of Computer Science, 1991.
28. Saraswat V.A., Concurrent Constraint Programming, MIT Press, Cambridge, 1993.
29. Shapiro E., The Family of Concurrent Logic Programming Languages, ACM Computing Surveys, 21(3):413-510, September 1989.
30. Shen K., The Extended CHR Implementation, chapter in ECL^{i}PS^e 4.0 Library Manual, IC-Parc at Imperial College, London, July 1998.
31. Sidebottom G.A., A Language for Optimizing Constraint Propagation, 1993, Simon Fraser University, Canada.
32. Smolka G, Treinen R.(Ed.), DFKI Oz Documentation Series, DFKI, Saarbrücken, Germany, 1994.
33. Ueda K., Chikayama T., Concurrent Prolog Compiler on Top of Prolog, in Symposium on Logic Programming, The Computer Society Press, pp.119-127, 1985.