Compiling Constraint Handling Rules

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

Abstract

We introduce the most recent and advanced implementation of CHR which improves both on previous implementations (in terms of completeness, flexibility and efficiency) and on the principles that should guide such an implementation. The idea is to have three rather independent phases of the compiler that utilize templates to generate the code and macros to specialize it. Moreover, our new implementation of CHR emphasizes the need for and power of attributed variables. We compile constraints into clauses and store them in attributes of variables. This is the first paper (besides from a lengthy technical report) that describes the implementation of CHR.

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 [D*88] allow the definition of propagation of constraints in a limited way.
- Constraint combinators in cc(FD) [VH91] allow to build more complex constraints from simpler constraints.
- Constraints connected to a Boolean variable in BNR-Prolog [BeOl92] and "nested constraints" [Sid93] allow to express any logical formula over primitive constraints.
- Indexicals in clp(FD) [CoDi93] allow to implement constraints over finite domains at a medium level of abstraction.

*The work was performed while visiting CWG at LMU with financial support from DFG.
†Constraint Working Group at Ludwig-Maximilians-University

- Meta- and attributed variables [Neu90, Hui90, Hol92] 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. Other (applications-specific) constraint domains can only be implemented using the last approach.

Attributed Variables [Hol92] serve as direct access storage locations for properties associated with variables. At the same time, attributed variables make the unification part of a unification based language, Prolog for example, user-definable within the language under extension [Hol90, Hol93]. 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 and ECL^{i}PS^e Prolog. However this 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) [Fru91, FrBr95a, FrBr95b, 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 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. This time difference 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 [D*88] constructs for user-defined constraints.

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. The probably most distinguishing functionality of CHR is that they act as a powerful iteration and retrieval mechanism over the constraint store, a data structure holding constraints.

CHR are typically realized as a library containing a compiler, runtime system and solvers written in CHR. For a CLP host language, the idea is to base the implementation of CHR on attributed variables. We will compile constraints into clauses and store them in attributes of variables. 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 which improves both on the previous implementation [FrBr95b] in terms of completeness, flexibility and efficiency and on the principles that should guide such an implementation [FrBr95a]. The improvements are based on compilation

on-the-fly and a special purpose CHR constraint debugger together with a rich but not overwhelming set of built-ins, options and pragmas. The release also includes more than 25 constraint solvers written in CHR.

The first implementation of CHR in 1991 was an interpreter written in ECL^{i}PS^e Prolog [Fru91]. Then, the CHR language has been implemented in 1993 in Common LISP at the German Research Institute for Artificial Intelligence (DFKI) [Her93] and in 1994 as a library of ECL^{i}PS^e [FrBr95a, FrBr95b]. CHR are currently also implemented in the successor language ECL^{i}PS^e 2 at IC-Parc of Imperial College and in the concurrent logical object-oriented constraint language OZ [SmTr94].

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 that will be based on attributed variables. We conclude with a comparison with the previous implementation. 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.

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* `primes(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.*

```
primes(1) <=> true.
generate @ primes(N) <=> N>1 | M is N-1, prime(N), primes(M).

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

Looking at the two rules defining `primes/1`*, note that head matching is used in CHR so the first rule will only apply to* `primes(1)`*. The test* `N>1` *is a guard (precondition) on the second rule named* `generate`*. Thus a call with a free variable, like* `primes(X)`*, will suspend. The third, multi-headed rule named* `absorb` *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, VH91, Sar93, JaMa94]. 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*[1]

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

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

where the rule has an optional `Name` *(a Prolog term), 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.*

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.

A simpagation rule combines a simplification and a propagation rule. The rule `Heads1 \ Heads2 <=> Body` is equivalent to the simplification rule `Heads1, Heads2 <=> Body, Heads1`. However, the simpagation rule is more compact to write, more efficient to execute and has better termination behaviour than the corresponding simplification rule.

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 as data in the constraint store. Every time a CHR constraint is executed (called) or woken (reconsidered), it checks itself the applicability of its associated CHR. 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*. 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.

The guard either succeeds or fails. A guard succeeds if the execution succeeds without *touching* a variable that occurs also in the heads. A variable is *touched* if it is unified with a non-variable term or a variable appearing in a CHR constraint or if it is the cause of an instantiation error. If the guard succeeds, the rule applies. Otherwise the next rule is tried.

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 with the same constraints if the constraint is woken. Since the currently active constraint has not been removed, the next rule is tried.

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. When a constraint is woken, all its rules are tried again.

[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.

3 The Compiler

The compiler is written in (SICStus) Prolog [HoFr98] and translates CHR into Prolog on-the-fly, while the file is consulted or compiled. Its kernel consists of a definite clause grammar that generates the target instructions (clauses) driven by templates. We will use example 1.1 to step through the three phases of the compiler: Parsing, translating CHR into clauses using templates and partial evaluation using macros.

Similar translations, i.e. compilation of committed-choice languages into Prolog, have been investigated before, be it translating GHC [UeCh85], implementations of delay declarations [Nai85] or the efficient implementation of QD-Janus [Deb93]. Today, we benefit from more powerful programming constructs, in particular customizable suspension mechanisms provided by attributed variables. The remaining issues to be addressed with CHR are multiple head constraints and propagation rules.

3.1 Parsing

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. The translation process requires mainly two synthesized attributes (pieces of information) to be computed from the parse tree:

- The set of global variables, i.e. those that appear in the heads of a rule.
- The set of variables shared between the heads matched so far and the current head.

In the canonical form of the rules,

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

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

Example 3.1 (Primes, contd.) *The canonical form of the rules for the prime number example is given below.*

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

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

3.2 Translating CHR into Clauses

Basically, CHR constraints are compiled into Prolog predicates. The CHR compiler associates each CHR constraint with all rules in whose heads it occurs. We first illustrate the compilation with a simple example, a single-headed simplification CHR, then we consider general cases of arbitrary multi-headed rules.

Each occurrence of a CHR constraint in the head of a rule gives rise to one clause. The clause head contains the active constraint, while the missing other head constraints are searched for in the constraint store.

Example 3.2 (Primes, contd.) *For the constraint* `primes/1` *the compiler generates the following intermediate code (edited for readability).*

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

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

% in chr primes(N) <=> N>1 | M is N-1, prime(N), primes(M)
primes(A) :-                            %  6
        match([C], [A]),                %  7
        check_guard([C], C>1),          %  8
        !,                              %  9
        D is C-1,                       % 10
        prime(C),                       % 11
        primes(D).                      % 12

% if no rule applied, suspend the constraint on its variables
primes_1(A) :-                          % 13
        suspend(primes(A)).             % 14
```

The predicate `match(L1,L2)` *matches the actual arguments (list* `L2`*) against the formal parameters (list* `L1`*). The predicate* `check_guard(L,G)` *checks the guard* `G`*. 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. Therefore,* `check_guard(L,G)` *fails as soon as the global variables (list* `L`*) are touched, i.e. takes part in a unification or gets more constrained by a built-in constraint.*

When no rule applied, the last clause inserts the constraint into the constraint store using a suspension mechanism. It allocates the suspension data structure(s) and attaches it to each variable occurring in the call. 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 crossproduct of matching constraints. For each tuple in the crossproduct the guard is checked in the corresponding environment. If the guard is satisfied, constraints that matched heads from 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 crossproduct.

Our implementation idea (as in [FrBr95a]) is an extension of the simple case presented above. For each head constraint the compiler does the following: It is deleted from the `Keep` or `Remove` list, respectively, and it will be called *active*. Whether the active constraint is removed when the rule applies, and whether any head constraints are removed, leads to the following three prototypical cases, each covered by a code generating template in the compiler. Interestingly, the three cases do not directly correspond to the three kinds of CHR.

Case `Active constraint from Remove list`

The active head constraint is to be removed if the rule applies. Thus the rule under consideration is either a simplification or simpagation rule. It can obviously be applied at most once with the current active constraint. The search for the partner constraints is through nondeterministic enumeration. Here is the template, slightly abridged. The predicate `ndmpc` generates the code to nondeterministically enumerate and match the partners, only by one.

```
compile(remove(Active), Remove, Keep, Guard, Body, ...) -->
        {
          Active =.. [_|Args],
          same_length(Args, Actual),
          ...
          ndmpc(Remove, RemoveCode, Ks, ...),
          ndmpc(Keep, KeepCode, ...)
        },
        [('F'(n(F/A,R-N), a(Actual)) :-
                  match(Args, Actual),
                  RemoveCode,
                  KeepCode,
                  check_guard(Vars, Guard),
                  !,
                  remove_constraints(Ks),
                  Body
        )].
```

Example 3.3 (Primes, contd.) *The second occurrence of prime/1 in rule 3 (Example 1.1) matches this template, and here is its instantiation:*

```
'F'(n(prime/1,3-2), a([A])) :-
        match([C], [A]),
    % RemoveCode (for one partner constraint)
        get_cnstr_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_cnstr_via(L,Cs)` *returns the constraints suspended on a free variable occurring in the list* `L`*. If there is no such variable, it returns all the constraints in the store.* `nd_init_iteration(Constraints, F/A, Candidate)` *nondeterministically returns a candidate constraint with functor* `F` *and arity* `A` *from the constraint store.*

Case `Active constraint from Keep list, Remove list nonempty`

This case applies only if there is at least one constraint to be removed and the active constraint will be kept. Thus the rule from which this case originates can

only be a simpagation rule. Since the active constraint is kept, one has to continue looking for applicable rules, even if the rule applies. 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. Therefore, we can deterministically iterate over the constraints that are candidates for matching this head, 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 3.4 (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 constraint predicate already flattened, as described in the next section on partial evaluation):*

```
prime(A, B) :-
        get_cnstr_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_2(C, B).                    % try next rule
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
```

Case `Active constraint from Keep list, Remove list empty`

In this case (stemming from propagation rules), there is no constraint to remove. 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 - this can be quite expensive.

Example 3.5 *This propagation rule is part of an interval solver.* `X::Min:Max` *constrains* `X` *with lower and upper bounds* `Min, Max`.

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

The propagation rule produces approximately the following code for `X le Y`. *Some arguments pushed around by the actual code have been omitted for readability.*

```
X le Y :- 'le/2_1'(X, Y).

'le/2_1'(X, Y) :-                           % active constraint X le Y
        get_cnstr_via([X], ViaX),
        init_iteration(ViaX, ::/2, Lx),
        !,
        'le/2_1_0'(Lx, X, Y).
```

```
'le/2_1'(X, Y) :-
        'le/2_2'(X, Y).

'le/2_2'(X, Y) :- suspend(X le Y).

'le/2_1_0'(Lx, X, Y) :-                     % outer loop for X::MinX:MaxX
        iteration_last(Lx),
        'le/2_2'(X, Y).
'le/2_1_0'(Lx, X, Y) :-
        iteration_next(Lx, Cx, LxRest),
        (   get_args(Cx), match,
            get_cnstr_via([Y], ViaY),
            init_iteration(ViaY, ::/2, Ly) ->
              'le/2_1_1'(Ly, LxRest, X, Y)
        ;
              'le/2_1_0'(LxRest, X, Y)
        ).

'le/2_1_1'(Ly, Lx, X, Y) :-                 % inner loop for Y::MinY:MaxY
        iteration_last(Ly),
        'le/2_1_0'(Lx, X, Y).
'le/2_1_1'(Ly, Lx, X, Y) :-
        iteration_next(Ly, Cy, LyRest),
        (   get_args(Cy), match ->
              Y::MinX:MaxY, X::MinX:MaxY,   % rule body
              'le/2_1_1'(LyRest, Lx, X, Y)
        ;
              'le/2_1_1'(LyRest, Lx, X, Y)
        ).
```

3.3 Partial Evaluation

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. [CaWi95]. In contrast to approaches that address all aspects of a language in a partial evaluator such as [Sah91], our restricted form of PE can be realized with an efficiency that meets the requirements of a production compiler.

The functionalities of the main macros of the compiler are:

- 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.
- Head matching is specialized into unification instructions guarded by `nonvar/1` tests (as in [UeCh85]).
- Structures are flattened by removing redundant function symbols. In particular, clause heads are flattened to facilitate clause indexing. For example, `'F'(n(prime/1,3-2), a([A]))` will be transformed into `prime_1(A)`.

Example 3.6 (Primes, contd.) *The macro expansion phase results in the following code for our example 3.2. The code for matching and guard checking has been in-lined. Then, the resulting trivial matchings (line 7), guards (line 3) and bodies (line 5) have been removed.*

```
primes(A) :-                        %  1
        A==1,                       %  2
        !.                          %  4
primes(A) :-                        %  6
        nonvar(A),                  %  8
        A>1,                        %  8
        !,                          %  9
        B is A-1,                   % 10
        prime(A),                   % 11
        primes(B).                  % 12
primes(A) :-                        % 13
        suspend(primes_1(A)).       % 14
```

4 The Runtime System

The code generated by the compiler utilizes Prolog directly 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

Suspended goals are our means to store constraints. They are re-executed each time one of their variables takes part in a unification and gets bound - either to a term or another variable associated with a suspension. This association can be realized at the Prolog source level via the attributed variables interface as found in SICStus or ECL^{i}PS^e Prolog, where the behaviour of attributed variables under unification is specified with a user-provided predicate.

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

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

The state basically indicates if the constraint is active, matched, removed or passive. The unique identifier is used, together with the propagation history, to ensure termination for propagation rules. Each propagation rule fires exactly once for each tuple formed by the set of matched head constraints. The tuples are stored in the propagation history of the involved constraints, i.e. suspensions. The re-use counter is incremented with every re-use of the suspension. It is used for profiling and some subtle aspects of rule application control outside the scope of this paper.

To make re-suspensions work as described, we made the suspension itself an argument of the re-executed goal. The user originally calls a constraint `c/n` as an ordinary goal. `c/n` calls `c/n+1` with an anonymous variable for the additional argument. `c/n+1` is the code for constraint `c/n` as generated by the CHR compiler. If the constraint has to delay, this extra argument is bound to the suspension. When it runs again, the suspension mechanism has a handle to the suspension and can update its state. This construction was omitted from the listed code samples in this paper to avoid clutter and puzzle.

4.2 Access Paths

A variable common to two heads of a rule considerably restricts the number of candidate constraints to be considered, because both partners must be suspended on this variable. Thus we usually access the constraint store by looking at only those constraints (cf. `get_cnstr_via/2`).

When a CHR searches for a partner constraint, we know functor and arity of the partner. Consequently, we want direct access to the set of constraints of the same functor/arity. Earlier implementations performed this selection by linear search over a part of the suspended constraints. In our runtime system we map every functor/arity pair to a fixed attribute slot at compile time yielding *constant time* access to the constraints of one type. Only the arguments need to be matched at runtime.

Access to data through a variable, and then functor/arity, is exactly the functionality provided efficiently by attributed variables. We store constraints directly in the attributes of variables. Typically, only very few constraints of a particular type will be suspended on a variable. Thus the constraint store can often be searched in constant time.

5 Empirics

We compared our new SICStus CHR [HoFr98] implementation with the one in distribution with ECL^{i}PS^e 3.5.2. We are measuring the variation between the two Prolog implementations together with the actual CHR implementation differences. The constraint solvers and examples are taken from the ECL^{i}PS^e distribution. 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 (with the `nodbgcomp` option). We have two columns for SICStus: one for native code, one for emulated code. We only compare SICStus emulated against ECL^{i}PS^e.

The new CHR version is 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 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).

Benchmark	sicstus native	a) sicstus emulated	b) eclipse	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
2nd solution	2.940	3.660	4.717	0.78

Most problems are well-known problems 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 reported by Bart Demoen of unknown origin. The word problem was provided by Klaus Schulz.

6 Conclusions

The CHR system outlined in this paper was implemented in four man-months. The compiler is 1100 lines of Prolog, the runtime system around 600, which is less than half of the ECL^{i}PS^e implementation.

Our new implementation rectifies some former limitations[3]:

- The CHR compiler has been "orthogonalized" by introducing three clearly defined compilation phases. Compilation is now on-the-fly. The template-based translation with subsequent macro-based partial evaluation allows for easy experimentation with different translation schemes. This led to a rather quick implementation of various compiler options and pragmas.
- 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 be provided, causing a significant speedup when compared to previous implementations.
- The number of heads in a rule is no longer limited to two. The restriction was motivated 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* [Sar93]. 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.)
- The runtime system no longer abuse the native suspension mechanism of the host language. The CHR specific demands, access paths and suspension recycling, are taken care of explicitly through customized versions of the suspension mechanism.

[3] Due to space limitations, only some of them have been discussed in this paper.

- The CHR debugging mechanism works by instrumenting the code generated by the CHR compiler. Basically, the CHR debugger works like the Prolog debugger, but there are extra ports specific to CHR. The entities reflected by the CHR debugger are constraints and rules. Constraints are treated like ordinary Prolog goals with the additional ports if they get inserted into or removed from the constraint store or if they are woken and reconsidered. Rules come with a *Try* and an *Apply* port. In addition, the constraint store and the ancestors of a constraint (the stack of calls leading to its execution) can be inspected.

Due to space limitations we 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.

Plans for the future development of the CHR implementation are the introduction of a priority scheme, realized through a scheduler [UeCh85] that fixes the remaining degrees of freedom with respect to sequencing[4], and the factorization of common matching instructions [Deb92].

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

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.

[BeOl92] F. Benhamou and W.J. Older, Bell Northern Research, June 1992, Applying interval arithmetic to Integer and Boolean constraints, Technical Report.

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

[CoDi93] 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.

[D*88] M. Dincbas et al., The Constraint Logic Programming Language CHIP, Fifth Generation Computer Systems, Tokyo, Japan, December 1988.

[Deb92] 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.

[Deb93] S. K. Debray, QD-Janus : A Sequential Implementation of Janus in Prolog, Software—Practice and Experience, Volume 23, Number 12, December 1993, pp. 1337-1360.

[FrBr95a] T. Frühwirth and P. Brisset, High-Level Implementations of Constraint Handling Rules, Technical Report ECRC-95-20, ECRC Munich, Germany, June 1995.

[FrBr95b] T. Frühwirth and P. Brisset, Chapter on Constraint Handling Rules, in ECL^{i}PS^e 3.5.1 Extensions User Manual, ECRC Munich, Germany, December 1995.

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

[4] e.g. the order in which applicable rules are executed

[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.

[Her93] B. Herbig, Eine homogene Implementierungsebene für einen hybriden Wissensrepräsentationsformalismus, Master Thesis, in German, University of Kaiserslautern, Germany, April 1993.

[Hol90] Holzbaur C, Specification of Constraint Based Inference Mechanisms through Extended Unification, Department of Medical Cybernetics and Artificial Intelligence, University of Vienna, Dissertation, 1990.

[Hol92] C. Holzbaur, 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.

[Hol93] C. Holzbaur, 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.

[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.

[Hui90] 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.

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

[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.

[Nai85] L. Naish, Prolog control rules, Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, California, September 1985, pp. 720-722.

[Neu90] U. Neumerkel, Extensible unification by metastructures, In Proc. of Metaprogramming in Logic (META'90), Leuven, Belgium, 1990.

[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.

[Sid93] G.A. Sidebottom, A Language for Optimizing Constraint Propagation, 1993, Simon Fraser University, Canada.

[SmTr94] G. Smolka and R. Treinen (Ed.), DFKI Oz Documentation Series, DFKI, Saarbrücken, Germany, 1994.

[UeCh85] Ueda K., Chikayama T, Concurrent Prolog Compiler on Top of Prolog, in Symposium on Logic Programming, The Computer Society Press, pp.119-127, 1985.

[VH91] P. Van Hentenryck, Constraint Logic Programming, The Knowledge Engineering Review, Vol 6:3, 1991, pp 151-194.