Translating Constraint Handling Rules into Action Rules

Tom Schrijvers[2⋆], Neng-Fa Zhou[1] and Bart Demoen[2]

[1] Department of Computer & Information Science
CUNY Brookly College & Graduate Center
zhou@csi.brooklyn.cuny.edu
[2] Department of Computer Science
K.U.Leuven, Belgium
{toms,bmd}@cs.kuleuven.be

Abstract. CHR is a popular high-level language for implementing constraint solvers and other general purpose applications. It has a well-established operational semantics and quite a number of different implementations, prominently in Prolog. However, there is still much room for exploring the compilation of CHR to Prolog. Nearly all implementations rely on attributed variables. In this paper, we explore a different implementation target for CHR: B-Prolog's Action Rules (ARs). As a rule-based language, it is a good match for particular aspects of CHR. However, the strict adherence to CHR's refined operational semantics poses some difficulty. We report on our work in progress: a novel compilation schema, required changes to the AR language and the preliminary benchmarks and experiences.

1 Introduction

Constraint Handling Rules (CHR) [8] is a rule-based programming language commonly embedded in a host language. It is a powerful yet relatively simple programming language that combines elements of Constraint (Logic) Programming (CLP) and rule-based languages. CHR's is intended as a language for implementing user-defined application-tailored constraint solvers, but it is also used as a general programming language.

Several implementations of CHR exist and most are embedded in Prolog [10, 12, 15] and HAL [5, 11]. CHR implementations for Java [1, 18] and Haskell [17] exist as well. The implementation [10] in SICStus Prolog is generally considered the reference implementation because it is historically the first full-fledged CHR system. It implements an efficient compilation schema in terms of attributed variables. More recently, the formulation of the refined operational semantics of CHR [6] has captured the essentials of the reference implementation on a more formal level.

AR (Action Rules) is a rule-based event-handling language originally designed for programming constraint propagation [19]. It has been employed in

⋆ Research Assistant of the fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

the implementations of several highly efficient constraint solvers. There are two major differences between AR and CHR: (1) CHR allows multi-headed rules while AR only accepts single-headed rules; and (2) unlike in CHR where primitive delay conditions are hidden in heads and guards of rules, all primitive delay conditions must be explicit in AR. It is not obvious how CHR can be compiled into AR, a seemingly lower-level language.

The goal of this paper is to explore a novel Prolog implementation of CHR in terms of Action Rules. Traditionally, CHR is compiled to Prolog using attributed variables [9], a low-level primitive for implementing constraint solvers. We argue that because of its rule-based nature, higher level of expressivity, Action Rules may provide a fair alternative compilation target. In our exploration we hope to assess this claim and gain new insights into the efficient compilation of CHR.

The rest of this paper is structured as follows. In the next section, we provide a brief overview of CHR-related concepts that are important for this paper. Subsequently, in Section 3 we introduce the Action Rules language. The novel compilation schema from CHR to AR is presented in two steps. Firstly, in Section 4, as basic compilation schema is presented. Secondly, a number of optimizations to this basic schema are sketched in Section 5. We report on a preliminary experimental evaluation in Section 6. Finally, in Section 7 we summarize and discuss our experiences.

There is a companion technical report [16] for this paper, that contains more elaborate explanation, full compilation schemas and optimizations, which we could not include in this text for lack of space.

2 Constraint Handling Rules

We assume the reader to be already familiar with the general aspects of the CHR language, its syntax and operational semantics.

The compilation schema we will present, implements the refined operational semantics [6], which is the de facto standard operational semantics of CHR implemented by all major CHR systems.

An important notation in the refined semantics, is the *occurrence* of a constraint symbol and the order of occurrences. The order is textual following the rules, and in a head it is from left to right, except for the simpagation rule, where occurrences to the right of the \ comes before the occurrences to the left.

The refined semantics follows a depth first execution strategy. One constraint is activated at a time and for this constraint the occurrences are visited in order. At each occurrence, the corresponding rule is attempted. If the rule succeeds, execution of the active constraint is temporarily suspended, in favor of depth first execution of the rule body. For further details we refer to [6].

3 AR: Action Rules

The AR (*Action Rules*) language is designed to facilitate the specification of event-driven functionality needed by applications such as constraint propagators and graphical user interfaces where interactions of multiple entities are essential [19]. It has been used to implement the efficient finite-domain and finite-set domain solvers in B-Prolog.

An action rule takes the following form: "$H, G, \{E\} \Rightarrow B$" where H (called the *head*) is an atomic formula that represents a pattern for agents, G (called the *guard*) is a conjunction of conditions on the agents, E (called *event patterns*) is a non-empty disjunction of patterns for events that can activate the agents, and B (called *action*) is a sequence of arbitrary subgoals. An action rule degenerates into a *commitment rule* if E together with the enclosing braces are missing. In general, a predicate can be defined with multiple action rules. For the sake of simplicity, we assume in this paper that each predicate is defined with only one action rule possibly followed by a sequence of commitment rules.

For this paper, we are interested in the following event patterns:

- `generated`: After an agent is generated but before it is suspended for the first time. The sole purpose of this event pattern is to make it possible to specify preprocessing and constraint propagation actions in one rule.
- `ins`(X): when the variable X is instantiated.
- `event`(X, T): the general form of user-defined events, where X is a variable, called a *channel*, and T is a Prolog term, called an *event object*.

In general, an action rule may specify several event patterns such as `ins` patterns on several variables.

For an agent, a rule is *applicable* to it if the agent matches the head of the rule and the guard is satisfied. When an agent is created, the system checks if the action rule in its predicate is applicable to it.[3] If so, the agent will be suspended until it is *activated* by one of the events specified in the rule. Whenever the agent is activated by an event, the condition of the action rule is tested *again*. If it is met, the action is executed. The agent does not vanish after the action is executed, but instead sleeps until it is activated again. There is no primitive for killing agents explicitly. An agent vanishes only when a commitment rule is applied to it.

Events are posted though a channel to agents by the system or by the user program. The built-in primitive `post_event`(C, T) posts a general event to the agents connected to C, where C is a *channel*.The activated agents are first connected to the active chain of agents and are then executed one at a time. Therefore, agents are activated in a *breadth-first* fasion. The second argument can be omitted if no information needs to be transmitted to the activated agents.

There is no primitive for killing agents explicitly. As described above, an agent never disappears as long as action rules are applied to it. An agent vanishes only when a commitment rule is applied to it. Consider the following example.

```
p(X,Flag), var(Flag), {event(X,O)} =>
         write(O),Flag=1.
p(X,Flag) => true.
```

[3] Since one-directional matching rather than full-unification is used to search for an applicable rule and only tests are allowed in the guard, the agent will remain the same after an applicable rule is found.

An agent defined here can only handle one event posting. After it handles an event, it binds the variable `Flag`. So, when a second event is posted, the action rule is no longer applicable and thus the commitment rule after it will be selected.

One question arises here: what happens if there will never be another event on `X`? In this case, the agent will stay forever. If we want to kill the agent immediately after it is activated once, we have to define it as follows:

```
p(X,Flag), var(Flag), {event(X,O),ins(Flag)} =>
        write(O),Flag=1.
p(X,Flag) => true.
```

In this way, the agent will be activated again after `Flag` is bound to 1, and be killed after the failure of the test `var(Flag)`.

4 The Basic Translation Schema

This section gives the general schema for translating CHR rules into AR rules. We focus on translating double-headed CHR rules. For single headed rules, a head constraint is treated as the partner of itself (discussed in Subsection 4.4), and rules with more than two constraint patterns in the head are translated to double-headed rule (discussed in Subsection 4.5) before translating them into AR rules.

4.1 Preparation

Without loss of generality we assume that all rules in the given CHR program have been transformed into the *head normal form* [11] where all matching operations are encoded explicitly in the guards of the rules. Hence, all the arguments of the constraint patterns in the program are unique variables.

For each constraint $p(\overline{X})$, an event of the following form is posted when the constraint is added into the store and when any variable in it is instantiated:

$$\texttt{constr}(Cno, Alive, History, \overline{X})$$

where Cno is a unique identifier of the constraint, $Alive$ is a logic variable, called a *status variable*, used to indicate aliveness of the constraint (the variable is free if the constraint is alive and is bound to 0 if it has been removed from the store), and $History$ is the firing history of propagation rules in which the constraint is involved. The data structure for the history is immaterial at this moment: it could very well be an open-ended list. We need the following primitives:

- `gen_constr_id`(Cno): Generates a new identifier Cno. The nature of the identifier is immaterial; an integer is used in our implementation.
- `not_in_history`($History, PartnerNo, RuleNo$):
- `add_to_history`($History, PartnerNo, RuleNo$): Let $History$ be the history associted with a constraint. The primitive `not_in_history` succeeds if the tuple $(PartnerNo, RuleNo)$ is not included in $History$, and the primitive `add_to_history` adds the tuple $(PartnerNo, RuleNo)$ into $History$.

For each occurrence of a constraint symbol in the heads, we use a unique identifier, called an *occurrence identifier*, to denote it, which is composed of the constraint symbol, the arity, and the number of the occurrence. For example. the fourth occurrence of a constraint symbol `p/1` has identifier `p_1_4`.

For each constraint symbol, we define two lists of occurrence identifiers: The *primal occurrence* list contains the occurrence identifiers of that symbol and the *partner occurrence* list contains the identifiers of the partner occurrences. The two lists overlap if there is a rule with two identical constraint symbols in the head. The ordering of primal occurrences is not important, but the occurrence identifiers in the partner list must be in the order specified by the refined operational semantics.

For each occurrence, a channel is used for posting and receiving events. For each instance in the store of the corresponding constraint symbol, there is an agent attached to the channel. We need the following primitive:

- $\texttt{get\_channel}(Id, Channel)$: retrieves the channel for the occurrence with the identifier `Id`.[4]

A channel is assigned to an identifier when `get_channel` is called on it for the first time.

4.2 The basic schema for two-headed rules

There is one Prolog clause generated for each constraint symbol. Let p/n be a constraint symbol with the primal occurrence list $[P_1, \ldots, P_k]$ and the partner occurrence list $[Q_1, \ldots, Q_m]$. Our basic schema generates the following Prolog clause for the symbol:

$$
\begin{array}{l}
p(\overline{X}) :- \\
\quad \texttt{gen\_constr\_id}(Cno), \\
\quad Constr = constr(Cno, Alive, History, \overline{X}), \\
\quad \texttt{get\_channel}(P_1, ChP_1), \ldots, \texttt{get\_channel}(P_k, ChP_k), \\
\quad \texttt{get\_channel}(Q_1, ChQ_1), \ldots, \texttt{get\_channel}(Q_m, ChQ_m), \\
\quad \texttt{agent\_}P_1(ChP_1, Cno, Alive, History, \overline{X}), \\
\quad \ldots \\
\quad \texttt{agent\_}P_k(ChP_k, Cno, Alive, History, \overline{X}), \\
\quad \texttt{post\_}p\_n(ChQ_1, \ldots, ChQ_m, Alive, Constr, \overline{X}).
\end{array}
$$

For each occurrence identifier in the primal and partner occurrence lists, there is a call of `get_channel`, which gets the channel for the occurrence. For each primal occurrence P_i ($i = 1, \ldots, k$), there is a call named $\texttt{agent\_}P_i$, which creates an agent for waiting for future partner constraints. The last call in the clause $\texttt{post\_}p\_n$ posts the constraint $p(\overline{X})$ to the channels of the partner occurrences to initiate search for partners of $p(\overline{X})$.

---

[4] The channel of an occurrence is an attributed variable stored as a global heap variable. In B-Prolog, the built-in `global_heap_get(Name,Value)`, which is the same as `b_getval(Name,Value)` in h-Prolog and SWI-Prolog, is used to access global heap variables.

Let the identifier of the rule in which P_i occurs be $RuleId$, the guard and body of the rule be G and B, respectively. If the rule is a propagation rule, then the agent for the occurrence P_i is defined as follows:

agent_$P_i(Ch, Cno, Alive, History, \overline{X})$,
    var($Alive$),
    {event(Ch, Q),ins($Alive$)}
=>
    $Q = constr(CnoQ, AliveQ, HistoryQ, \overline{Y})$,
    (Cno\==$CnoQ$,
    var($AliveQ$)
     not_in_history($History, CnoQ, RuleId$),
     not_in_history($HistoryQ, Cno, RuleId$),
     G ->
         add_to_history($History, CnoQ, RuleId$),
         add_to_history($HistoryQ, Cno, RuleId$),
         B
     ;
         true).
agent_$P_i(_, _, _, _, \overline{X})$ =>true.

When the agent receives an event $constr(CnoQ, AliveQ, HistoryQ, \overline{Y})$, it checks the following: (1) the constraint represented by the event is different from the constraint represented by this agent (Cno\==$CnoQ$); (2) the constraint represented by the event is alive (var($AliveQ$)); (3) the rule $RuleId$ has never been applied on $(Cno, CnoQ)$ or $(CnoQ, Cno)$; and (4) the guard G is satisfied. The body is executed when all these four conditions are satisfied. The histories associated with these two constraints are updated to record this application before the body is executed. Notice that the agent also watches the ins($Alive$) event. When the constrain represented by the agent is removed from the store ($Alive$ is bound to 0), the agent will be killed.

If the rule in which P_i occurs is a simplification rule, then the following action rule is generated for P_i:

agent_$P_i(Ch, Cno, Alive, History, \overline{X})$,
    var($Alive$),
    {event(Ch, Q),ins($Alive$)}
=>
    $Q = constr(CnoQ, AliveQ, HistoryQ, \overline{Y})$,
    (Cno\==$CnoQ$,
    var($AliveQ$)
     G ->
         $Alive = 0, AliveQ = 0$,
         B
     ;
         true).

The status variables *Alive* and *AliveQ* are set to 0 to indicate that the constraints represented by the agent and the event have been removed from the store. If the rule is a simpagation rule, then only *Alive* or *AliveQ* is set to 0 depending on whether P_i occurs to the left or right of \.

After all the agents `agent_`P_1, ..., `agent_`P_k are created, an agent named `post_`p_n is created to initiate search for partners.

$$
\begin{array}{l}
\texttt{post\_}p\_n(ChQ_1,\ldots,ChQ_m,Alive,Constr,\overline{X}) \\
\quad \texttt{var}(Alive), \\
\quad \{\texttt{generated},\texttt{ins}(Alive),\texttt{ins}(\overline{X})\} \\
=> \\
\quad \texttt{post\_event}(ChQ_1,Constr), \\
\quad \ldots \\
\quad \texttt{post\_event}(ChQ_m,Constr). \\
\texttt{post\_}p\_n(ChQ_1,\ldots,ChQ_m,\_,\_,\overline{X}) => \texttt{true}.
\end{array}
$$

When the constraint $p(\overline{X})$ is created (as indicated by the event pattern `generated`) or when any variable in it is instantiated (as indicated by the event pattern `ins`$(\overline{X})$), the event representing the constraint *Constr* is posted to the channels of the partner occurrences to initiate search for partners. The refined semantics is preserved by posting the constraint to the channels of the occurrences in the specified order.

4.3 An Example

Consider the following CHR rule:

```
prime(Y) \ prime(X) <=> 0 is X mod Y | true.
```

where `prime(X)` is numbered 1 and `prime(Y)` is numbered 2. The following shows the generated code.

```
prime(X) :-
        gen_constr_id(Id),
        Constr = constr(Id,Alive,History,X),
        get_channel(prime_1_1,Ch1),
        get_channel(prime_1_2,Ch2),
        agent_prime_1_1(Ch1,Id,Alive,History,X),
        agent_prime_1_2(Ch2,Id,Alive,History,X),
        post_prime_1(Ch1,Ch2,Alive,Constr,X).

agent_prime_1_1(Ch,Id,Alive,History,X), var(Alive),
        {event(Ch,Constr), ins(Alive)} =>
        Constr=constr(IdQ,AliveQ,HistoryQ,Y),
        (Id\==IdQ,
         var(AliveQ),
         0 is X mod Y ->
            Alive = 0
```

```
        ;
            true
        ).
agent_prime_1_1(_,_,_,_,_) => true.

agent_prime_1_2(Ch,Id,Alive,History,Y), var(Alive),
        {event(Ch,Constr), ins(Alive)} =>
        Constr=constr(IdQ,AliveQ,HistoryQ,X),
        (Id\==IdQ,
         var(AliveQ),
         0 is X mod Y ->
            AliveQ = 0
        ;
            true
        ).
agent_prime_1_2(_,_,_,_,_) => true.

post_prime_1(Ch1,Ch2,Alive,Constr,X), var(Alive),
        {generated, ins(Alive), ins(X)} =>
        post_event(Ch1,Constr),
        post_event(Ch2,Constr).
post_prime_1(_,_,_,_,_) => true.
```

4.4 Single-headed rules

For a single-headed CHR rule with the constraint symbol p/n in the head, a constraint of p/n does not need to wait for a partner constraint to trigger the rule. Let P_i be the occurrence identifier of the head. Each time a constraint of p/n is added into the store, an agent defined below is created:

$\texttt{agent\_}P_i(PrivateCh, Cno, Alive, History, \overline{X}),$
$\quad \texttt{var}(Alive),$
$\quad \{\texttt{event}(PrivateCh, \_),\texttt{ins}(Alive)\}$
$=> \ldots$
$\texttt{agent\_}P_i(\_, \_, \_, \_, \overline{X}) => \texttt{true}.$

where the body of the action rule encodes the guard and body of the CHR rule. The agent is not activated by the posting of a partner constraint, but by the posting of the constraint itself. The event pattern $\texttt{event}(PrivateCh, \_)$ means that no information from any event is used. By using a separate channel for each occurrence of p/n, we enforce as before the ordering of the rules in the refined semantics. However, now we have to use a private channel (one for each constraint instance rather than a global one) to enforce the depth-first execution order of the refined semantics. Otherwise we would break our assumption that a constraint only features as its own partner for a single-headed rule. The result would be a kind of breadth-first execution order.

Consider the following CHR program, (partially) implementing boolean `and/3` and `not/2` constraints:

```
and(X,X,Z) <=> Z = X.
not(X,Y) \ and(X,Y,Z) <=> Z = 0.
```

The following shows the generated code for `and/3`.

```
and(X,Y,Z) :-
        gen_constr_id(Cno),
        Constr = constr(Cno,Alive,History,X,Y,Z),
        get_channel(and_3_2,PublicCh2),
        get_channel(not_2_1,ChNot),
        agent_and_3_1(PrivateCh1,Cno,Alive,History,X,Y,Z),
        agent_and_3_2(PublicCh2,Cno,Alive,History,X,Y,Z),
        post_and_3(PrivateCh1,ChNot,Alive,Constr,X,Y,Z).

agent_and_3_1(PrivateCh1,Cno,Alive,History,X,Y,Z), var(Alive),
        {event(PrivateCh1,_), ins(Alive)} =>
        (X == Y ->
            Alive = 0,
            Z = X
        ;
            true
        ).
agent_and_3_1(_,_,_,_,_,_,_) => true.

agent_and_3_2(PublicCh2,Cno,Alive,History,X,Y,Z), var(Alive),
        {event(PublicCh2,Constr), ins(Alive)} =>
        Constr=constr(Cno,AliveNot,HistoryNot,XNot,YNot),
        (Cno\==CnoNot,
         var(AliveNot),
         X == XNot,
         Y == YNot
          ->
            Alive = 0,
            Z = 0
        ;
            true
        ).
agent_and_3_2(_,_,_,_,_,_,_) => true.

post_and_3(PrivateCh1,ChNot,Alive,Constr,X,Y,Z), var(Alive),
        {generated, ins(Alive), ins(X), ins(Y), ins(Z)} =>
        post_event(PrivateCh1,Constr),
        post_event(ChNot,Constr).
post_and_3(_,_,_,_,_,_,_) => true.
```

4.5 Multi-headed rules

For rules with more than two constraint patterns in the heads, we apply a similar binarization technique as in the famous RETE algorithm [7] for production systems. This technique makes the intermediate joins of partner constraints manifest in temporary constraints.

Conceptually for a rule with $n \geq 2$ constraint patterns:

$$p_1(\overline{X}_1), \ldots, p_n(\overline{X}_n) ==> G \mid B.$$

We compile it into $(n-1)$ two-headed rules with $(n-1)$ new constraint symbols $\mathtt{p}_{1\ldots j}$ where $2 \leq j \leq (n-1)$:

$$\begin{array}{l}
p_1(\overline{X}_1), p_2(\overline{X}_2) ==> p_{1..2}(\overline{X}_1, \overline{X}_2). \\
p_{1..2}(\overline{X}_1, \overline{X}_2), p_3(\overline{X}_3) ==> p_{1..3}(\overline{X}_1, \overline{X}_2, \overline{X}_3). \\
\vdots \\
p_{1..(n-1)}(\overline{X}_1, \ldots, \overline{X}_{(n-1)}), p_n(\overline{X}_n) ==> p_{1..n}(\overline{X}_1, \ldots, \overline{X}_n).
\end{array}$$

It has often been claimed informally that the correct binarization of CHR rules can be expressed trivially as the above source-to-source transformation is correct. In effect, early implementations of CHR did not support any n-headed rules where $n > 2$, partially for that reason.

In reality, the binarization is not as simple as that, and we believe that it cannot be expressed fully as a source-to-source transformation. A number of issues arise that have to be dealt with at a lower level.

Firstly, no head constraint can be used twice in the matching against the constraint patterns in the original CHR rule. To ensure this, we let each temporary constraint carry the identifiers of the precedence constraints from which it is obtained. Each time before we fire the rule $p_{1..j}(\overline{X}_1, \ldots, \overline{X}_j), p_{j+1}(\overline{X}_{j+1}) ==> \ldots$, we check if $p_{j+1}(\overline{X}_{j+1})$ occurs in the precedence constraints that led to the generation of $p_{1..j}(\overline{X}_1, \ldots, \overline{X}_j)$. If so, the rule cannot be fired.

Secondly, the following relationship among the status variables must be maintained: whenever a precedence constraint is removed from the store, the temporary constraints created from it must be removed as well. For this reason we let each temporary constraint carry in a similar fashion the status variables of the precedence constraints. On an `ins` event of any of these status variables the agents for the temporary constraints are removed.

Thirdly, for simplification and simpagation rules, it should be possible at the end of the join chain, before executing the body of the original rule, to remove the original precedence constraints. For this purpose, the status variables of the precedence constraints must be carried by the temporary constraints. Luckily, this is already the case for the previous issue.

Finally, the temporary constraints should be properly synchronized with their precedence constraints. If a variable appearing in a precedence constraint is instantiated, any temporary constraint that caries that variable should not fire its event, before the events for the previous occurrences of the precedence constraint do. Our solution is to not fire events for temporary constraints upon `ins` events of the temporary constraints' variables. Instead every precedence constraint has a private channel for each multi-headed occurrence. This channel is carried by the temporary constraints. Upon an `ins` event for the precedence constraint, first the usual events are posted for the preceding occurrences before an event is posted to this private channel. The temporary constraint react on this event by firing their usual events.

Consider the following artificial CHR program:

```
a(X) \ b(X), c(X) <=> X = 42.
```

The following shows the generated code for `a/1` and the temporary constraint `temp12/8`.

```
a(X) :-
        gen_constr_id(Cno),
        Constr = constr(Cno,Alive,History,X,ChTemp),
        get_channel(a_1_1,Ch1),
        get_channel(b_1_1,ChB),
        agent_a_1_1(Ch1,Cno,Alive,History,X,ChTemp),
        post_a_1(ChB,ChTemp,Alive,Constr,X,Y,Z).

agent_a_1_1(Ch,Cno,Alive,History,X,ChTemp), var(Alive),
        {event(Ch,Constr), ins(Alive)} =>
        Constr=constr(CnoB,AliveB,HistoryB,XB,ChTempB),
        (Cno\==CnoB,
         var(AliveB),
         not_in_history(History,CnoB,r1),
         not_in_history(HistoryB,Cno,r1)
          ->
            add_to_history(History,CnoB,r1),
            add_to_history(HistoryB,Cno,r1),
            temp12(X,XB,Cno,CnoB,Alive,AliveB,ChTemp,ChTempB)
        ;
            true
        ).
agent_a_1_1(_,_,_,_,_,_) => true.

post_a_1(ChB,ChTemp,Alive,Constr,X), var(Alive),
        {generated, ins(Alive), ins(X)} =>
        post_event(ChB,Constr),
        post_event(ChTemp,Constr).
post_a_1(_,_,_,_,_) => true.

temp12(XA,XB,CnoA,CnoB,AliveA,AliveB,ChTempA,ChTempB) :-
        gen_constr_id(Cno),
        Constr = constr(Cno,Alive,History,XA,XB,...),
        get_channel(temp12_1,Ch1),
        get_channel(c_1_1,ChC),
        agent_temp12_8_1(Ch1,Cno,Alive,History,XA,XB,...),
```

```
        post_temp12_8(ChC,Alive,Constr,AliveA,AliveB,ChTempA,ChTempB).

agent_temp12_8_1(Ch1,Cno,Alive,History,XA,XB,CnoA,CnoB,AliveA,AliveB,ChTempA,ChTempB),
        var(Alive),var(AliveA),var(AliveB),
        {ins(Alive),ins(AliveA),ins(AliveB),event(CH1,Constr)} =>
        Constr=constr(CnoC,AliveC,HistoryC,XC,ChTempC),
        (Cno\==CnoC, CnoA\==CnoC, CnoB\==CnoC,
         var(AliveB),
         XA==XB, XB==XC
         ->
            AliveB=0, AliveC=0,
            XA = 42
        ;
            true
        ).
agent_temp12_8_1(_,_,_,_,_,_,_,_,_,_,_,_) => true.

post_temp12_8(ChC,Alive,Constr,AliveA,AliveB,ChTempA,ChTempB),
        var(Alive),var(AliveA),var(AliveB),
        {generated,ins(Alive),ins(AliveA),ins(AliveB),
         event(ChTempA,ChTempB} =>
        post_event(ChC,Constr).
post_temp12_8(_,_,_,_,_,_,_) => true.
```

5 Optimizations

The above compilation schema is quite basic in a number of ways. This section discusses a few optimizations that greatly improve the efficiency. We focus on significant adaptations of existing optimizations [11, 14] for ARs.

Indexing. In our basic schema the public channels used for finding partner constraints have no selectivity at all: all constraints with the appropriate constraint symbol are reached. Indexing greatly reduces the selectivity of partner lookups. In our AR schema we associate an index with a particular partner constraint lookup. In addition to the general channel for the lookup, one or more index channels may be provided. An index channel only reaches a constraint that has a particular variable or a particular term in some argument position. The occurrence agent of the constraint simply watches the additional index channels too. The post agent updates the constraint's index channels upon instantiation, and it looks for partner constraints through index channels if they are available.

If multiple index channels are available, these are combined using the new `post_event(Index1/\.../\Indexn,Event)` feature, that only posts the event to agents that are attached to *all* of the channels `Index1,...,Indexn`.

Single-headed rules. We can merge the agents of multiple consecutive single-headed rules and their corresponding private channels into a single agent and a single private channel. The bodies of the rules are merged appropriately in a single body. Subsequent bodies of a propagation rule are put in conjunction (i.e. any continuation), and of simplification rules are put in the else-branch (i.e. fail continuation) of that body.

For a constraint symbol that is defined by only single-headed rules, no private channel is needed at all to control the ordering.

Never-triggering constraints. A constraint symbol whose instances do not *trigger*, e.g. because they are fully instantiated when called, does not require its agents to watch `ins/1` events of the arguments.

6 Evaluation

We have written an automatic translator from CHR to Action Rules, following the above basic compilation schema. This compiler is evaluated on a representative set of benchmarks from [13]. In addition, to get a good view on the attainable efficiency, we have further optimized the generated code of the smallest benchmarks by hand. The compiler, benchmarks and hand-optimize code are all available from the webpage `http://www.probp.com/chr/`.

As a reference for comparison, we take the K.U.Leuven CHR system [15] in SWI-Prolog. This is currently the most optimized CHR-system for Prolog. Tabel 1 lists the timings in milliseconds, obtained on a Pentium 4 2 GHz.

Benchmarks	B-Prolog		SWI-Prolog
	Basic Compiler	Hand-Optimized	K.U.Leuven CHR
`fib (22)`	38,333 (*)	246	3,210
`fulladder (6000)`	1,050	-	340
`leq (50)`	82,823	138	3,010
`leq2 (50)`	84,407	145	3,870
`primes (2500)`	3,049	1,350	6,990
`wfs (1000)`	11,783	-	2,920
`zebra (10)`	4,273	621	4,580

Table 1. Benchmark Timings

In order to interpret the figures in Tabel 1 correctly, one must take into account that B-Prolog is on average about 5 times faster than SWI-Prolog.

Unsurprisingly, the basic schema's performance is not so good. However, the hand-optimized code shows that drastic improvements are possible. In fact, such improvements are capable of considerably outperforming the K.U.Leuven CHR system in its current form. This suggests a worthwhile set of optimizations that can be generalized to other compilation schema, such as that of the K.U.Leuven CHR system.

In (*) we use the consulted rather than the compiled benchmark, because the garbage collector is not activated in the latter, causing a serious slowdown. We expect this to be fixed in a new release of B-Prolog. Furthermore, we cannot reliably experiment with multi-headed rules that involve **Reactivate** transitions yet. The Action Rules semantics currently allows for a delay between an instantiation and its corresponding `ins/1` event, where other code can be run in between that is out of order with respect to CHR's refined semantics. We will also address this issue in a future version.

7 Conclusion

In this paper we have presented a novel schema for compiling CHR to Action Rules. On the one hand, the AR schema has some elegance over the attributed variables schema in the expression as it allows some loops to be expressed implicitly, just like CHR. On the other hand, we have experienced some difficulty with strictly following the refined operational semantics of CHR; it is not such a good match for the operational semantics of AR.

Our initial experimental results show that the basic performance is rather bad, but hand-optimizations give very encouraging results and suggest worthwhile optimizations for all CHR systems. Another important result of our schema is that it shows how to properly binarize multi-headed rules, and that this is non-obvious. This shows the advantages of the attributed variables schema over the RETE-based approach. In [4] it was already shown that RETE's time and space behavior is worse than that of LEAPS [3], by which the attributed variables implementation of CHR was inspired.

A current limitation of the Action Rules schema is that it does not fully support an extension of CHR, called $\mathrm{CHR}^{\vee}$ [2], which allows disjunctions in the bodies of rules. An Action Rule behaves as if the body is contained inside the ISO-Prolog `once/1` built-in: after executing the body, any remaining choice points created during the execution are pruned.

In future work, we will extend and automate the proposed optimizations. In particular, we will focus on Action Rules specific optimizations. A hybrid compilation schema seems another worthwhile topic for further research: selecting the most efficient parts of the attributed variables and Action Rules schemas, possibly based on properties of the program being compiled. This could also allow us to lift the current restriction of the Action Rule schema and fully support $\mathrm{CHR}^{\vee}$. Finally, we will investigate were the strict adherence to the refined operational semantics can be lifted without harm.

References

1. Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauss. JACK: A Java Constraint Kit. In *Proceedings of the International Workshop on Functional and (Constraint) Logic Programming*, Kiel, Germany, September 2001.
2. Slim Abdennadher and Heribert Schütz. $\mathrm{CHR}^{\vee}$: a flexible query language. In *FQAS '98: Proceedings of the Third International Conference on Flexible Query Answering Systems*, pages 1–14, London, UK, 1998. Springer-Verlag.

3. Don Batory. The leaps algorithm. Technical report, Austin, TX, USA, 1994.
4. David A. Brant, Timothy Grose, Bernie Lofaso, and Daniel P. Miranker. Effects of database size on rule system performance: Five case studies. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 287–296. Morgan Kaufmann, 1991.
5. Bart Demoen, Maria García de la Banda, Warwick Harvey, Kim Marriott, and Peter J. Stuckey. An Overview of HAL. In Joxan Jaffar, editor, *CP'99: Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*, pages 174–188, Alexandria, Virginia, USA, 1999. Springer Verlag.
6. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *ICLP'04: Proceedings of the 20th International Conference on Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104, St-Malo, France, September 2004. Springer Verlag.
7. Charles L. Forgy. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19(1):17–37, 1982.
8. Thom Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
9. Christian Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. Technical Report TR-92-23, Austrian Research Institute for Artificial Intelligence, Vienna, Austria, 1992.
10. Christian Holzbaur and Thom Frühwirth. A Prolog Constraint Handling Rules Compiler and Runtime System. *Special Issue Journal of Applied Artificial Intelligence on Constraint Handling Rules*, 14(4), April 2000.
11. Christian Holzbaur, María García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing Compilation of Constraint Handling Rules in HAL. *Theory and Practice of Logic Programming: Special Issue on Constraint Handling Rules*, 5(Issue 4 & 5):503–531, 2005.
12. IC-Parc. $\mathrm{ECL}^i\mathrm{PS}^e$. http://www.icparc.ic.ac.uk/eclipse/.
13. Tom Schrijvers. A Collection of Assorted CHR Benchmarks, 2005. http://www.cs.kuleuven.be/~toms/Research/CHR/.
14. Tom Schrijvers. *Analyses, optimizations and extensions of Constraint Handling Rules*. Phd, Department of Computer Science, K.U.Leuven, Leuven, Belgium, June 2005. xxiv+210 pages.
15. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In Thom Frühwirth and Marc Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, pages 1–5, Ulm, Germany, May 2004.
16. Tom Schrijvers, Neng-Fa Zhou, and Bart Demoen. The compilation schema for translating chr into action rules. Report CW 449, K.U.Leuven, Department of Computer Science, June 2006.
17. Peter J. Stuckey and Martin Sulzmann. A Theory of Overloading. *ACM Transations on Programming Languages and Systems*, 2005. To appear.
18. Armin Wolf. Adaptive Constraint Handling with CHR in Java. In *CP'01: Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science 2239, page 256. Springer Verlag, January 2001.
19. Neng-Fa Zhou. Programming finite-domain constraint propagators in action rules. *To appear in Theory and Practice of Logic Programming (TPLP)*, 2006.