Reasoning about passive declarations in CHR

Henning Christiansen

Roskilde University, Computer Science Dept. DK-4000 Roskilde, Denmark
E-mail: `henning@ruc.dk`

Abstract. The programming language of Constraint Handling Rules (CHR) is gaining more and more popularity and this has motivated the development of new optimization techniques to be applied in implementations of CHR. As for other logic programming languages, a program written CHR can be understood declaratively as a logical formula and as a procedural specification, and CHR has different tools for manual optimization at the procedural level whose application may (or may not) sacrifice the declarative reading. One such optimization is given by passive declarations by means of which the search for rules to be applied can be optimized, perhaps changing which rules that are applied and in which order. A framework for analyzing the effect of passive declarations is presented in terms of a slightly abstract operational semantics which takes into account the effect of passive declarations. Based on it, we can give different classifications of the effect of some passive declarations that do not affect the logical semantics. This may be used for reasoning about manual optimizations as well as proposals for automatic strategies to be applied by a compiler for adding passive declarations.

1 Introduction

The programming language of Constraint Handling Rules [4] (CHR) was originally introduced as a specialized tool for writing constraint solvers for standard domains such as real and rational arithmetics, finite domains, etc., but has gained a growing popularity as a programming language of a wider scope. Implementations of CHR are now available in a number of Prolog systems (e.g., [14]) and other platforms,[1] and the development of program analysis and optimization methods for CHR is currently an active research area.

CHR includes passive declarations by means of which a programmer can affect the search for rules to be applied in order to obtain a faster program execution. Passive declarations may or may not affect the declarative semantics of a program, and we present an operational semantics of CHR that captures the effect of passive declarations, and by means of which we can characterize some classes of programs where the declarative semantics is preserved. This framework can be used for reasoning about manual optimizations as well as strategies to

[1] The web page http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/ refers at the time of writing to nine implementations embedded in Prolog, one in Haskell and one in Java.

be applied in CHR compilers or high-level source-to-source translators; as an example of the latter, we consider optimizations applied in the CHRG system [2] which translates a grammar notation into CHR.

Recently, other authors have proposed what they call a refined operation semantics [3] which is intended to capture all procedural aspects of actual CHR implementations, and as such it is completely deterministic. By nature, this refined semantics is very complicated and detailed, and for our own purposes, which includes doing proofs by hand, we have chosen a more abstract version which includes sufficient details to characterize essential properties of passive declarations, but is nondeterministic for the rest; we have, however, not made a formal comparison of the two semantics.

As will be shown below, passive declarations of a program in general can only preserve the semantics for a specific class of queries. The current CHR syntax does not recognize such restrictions to classes of queries and in this way, our methodology may in some cases go further than what is possible in existing compilers.

Over the last 50 years, the need for efficient compilers has driven the development of sophisticated methods for control and data flow analyses, including abstract interpretation, for other programming languages, including traditional logic programming languages such as Prolog. For CHR, on the other hand, such analyses tend to be quite difficult due to the demon-like nature of rules, and it is only very recently that such methods have been suggested; references are given below.

Based on our operational semantics, we define an important class of *control-safe* passive declarations that do not change the order in which rules are applied, but inform the CHR compiler where optimization can be made. We indicate also a wider class of *answer-safe* passive declarations which may change flow of control but do not change result of the computation.

In section 2 we review informally the operational semantics of CHR and discuss the meaning of passive declations, and the two notions of safety are defined; section 3 presents the operational semantics. In section 4, we present a method for proving control-safety based on local properties of each rule; at present it is not suited for automatic systems as it requires the invention of a state invariant. Classes of programs, for which we can point to patterns of passive declarations guaranteed to be control-safe, are given in section 5, CHRs for parsing and a class of programs with a separation of constraints into data structure and process constraints. Section 6 considers the answer-safe (and not necessarily control-safe) passive declarations for programs that have been proved to be confluent. Here a simple criterion based on subsumption of rule heads applies. Section 7 analyzes the speedup gained by passive declarations and provides a few benchmark tests. Sections 8 and 9 review briefly related works and give conclusion and perspectives.

2 CHR and passive declaration

For an introduction to CHR and its declarative semantics, we refer to [4]. Similarly to [3] we consider only simpagation rules, as they subsume both propagation and simplification rules. The general form is as follows; P, Q, and R are sequences of constraints, G a guard which is a test on variables of P and Q made by built-in test predicates. For simplicity, we allow only the built-in constraint "=" with the usual meaning of unification in R (but not in P and Q).[2]

$$P \setminus Q \quad \texttt{<=>} \quad G \mid R$$

The left hand side $P \setminus Q$ is called the *head* of the rule and R the *body*. In case P is empty, the backslash is left out and the rule is called a simplification rule. When Q is empty, we talk about a propagation rule, leaving out again the backslash, but replacing the arrow symbol with `==>`. Procedurally, a CHR program can be understood basically as the rewriting of constraint stores with the initial constraint store given by a query. A simpagation rule as above can be applied whenever constraints matching P, Q are found in the store and G is satisfied; in that case, constraints matched by Q are removed from the store and the corresponding instance of R is added. This principle can with a few simple adjustments be formalized into an operational semantics which satisfies standard correctness properties wrt. a declarative semantics. Unfortunately, the statements provided by the declarative semantics or the operational one indicated above, are often very weak, and it may be claimed that CHR programmers tend to work with the *de-facto* operational semantics given by the current implementations (e.g., the version in SICStus Prolog [14] considered the reference implementation) in mind rather that the logical one. As noted by [3], even example programs in the reference manual of CHR can only be explained by referring to the *de-facto* operational semantics. This motivated [3] to describe a so-called refined operational semantics that gives the precise control flow of the *de-facto* operational semantics. The following properties are essential in the refined semantics:

1. Calls of constraints are procedure calls. When a constraint c is called and it triggers some rule, then the constraints in that rule's body are called recursively, triggering other rules, and control can only return to c when these other rules have finished (unless c has been consumed by a simplification/simpagation step).
2. Left-to-right execution. The constraints in a query or rule body are called in a sequential manner from left to right.
3. When a constraint c is called, rules containing the same predicate as c are tested in their textual order for possible application.

[2] In versions of CHR embedded in Prolog. e.g., [14], the guard may also refer to user-defined predicates; we ignore the possible splitting of the guard into an ask and tell part. Typically, the body R can be any callable Prolog term which will make analysis more complicated but does not change the overall mechanisms.

4. When a constraint c is called, it gets the status of "operational" meaning that other rules triggered directly or indirectly by c can access c and perhaps even delete it by means of a simplification rule. (Intuitively, an operational constraints such as c belongs at this point to both the constraint store and to the recursion stack.)

Example 1. The last property can be illustrated by the following CHR program which has been tested in a running implementation of CHR.

```
a     ==>  write('rule1 '), b.
a, b <=>  write('rule2 '), c.
b     <=>  write('rule3 ').
```

A call to `a` results in the printout `rule1 rule2` and a final constraint store consisting of `c`. The call to `a` gives rise to a call to `b`; the first rule with `b` in the head is tried, and since `a` is accessible as "operational", this rule can fire, thus consuming `a` and `b` as it is a simplification rule.

Constraints in the head of a rule may be declared passive using the following syntax.

```
a(X)#Id, b(X) ==> c(X) pragma passive(Id)
```

The rule behaves like `a(X), b(X) ==> c(X)` except that, when a call `a(···)` tries all rules in textual order, this one is skipped. Thus this rule can only be triggered by a call of form `b(···)` and, of course, when a suitable operational `a(···)` can be located. It will be a straightforward matter to incorporate this behaviour in the refined semantics [3] which we shall refrain from here.

¿From now on we abbreviate the syntax for passive declarations to just the `#` symbol on the head constraints. To simplify notation, we indicate a set of passive declaration of a specific program as an operator $\mathcal{P}$, and when writing $\mathcal{P}(\Pi)$ for program Π, it is assumed that Π refers to the version without passive declarations; similarly, for a rule $\mathcal{P}(\rho)$ of $\mathcal{P}(\Pi)$, the argument ρ refers to the matching rule of Π. It is understood that rules and elements of rules occur in a specific order and that the elements of $\mathcal{P}$ refer to specific positions in rule heads.

Assume, now, that the sample rule above is part of a program which has the overall property that whenever a call `a(`t`)` is made operational during execution, there cannot be any matching `b(`t`)` operational in the same state. Then, clearly, the passive declaration does not affect the order in which the rules are applied and thus the final result of the computation. This is an instance of what we define below as control-safety. In general, however, such a properties can only be ensured under given restrictions to the input queries. If, for example, the query `b(1),a(1)` is possible, the desired property does not hold.

A semiformal definition is sufficient for our purposes, based on the understanding of an operational semantics being a device that takes a query (a sequence of constraints) plus program as input and generates a set of derivations being sequences of states with each transition indicated by the instance of the specific rule being applied. A derivation has an initial state isomorphic to a query, and may be infinite or have a final state from which a computed answer is abstracted.

Definition 1. *Assume an operational semantics for CHR, a program $\mathcal{P}(\Pi)$ and a set of queries Θ. Then the passive declarations $\mathcal{P}$ of $\mathcal{P}(\Pi)$ are said to be*

- control-safe *for Θ whenever the set of derivations for queries in Θ is independent of whether program Π or $\mathcal{P}(\Pi)$ is used, and*
- answer-safe *for Θ whenever the set of pairs*
  $\{\langle Q, \textit{computed answer for } Q\rangle \mid Q \in \Theta\}$
  is independent of whether program Π or $\mathcal{P}(\Pi)$ is used.

Control-safety implies answer-safety but not vice versa. For simplicity, we do not consider the case where the user has supplied some passive declarations already before analysis takes place. It is, however, straightforward to generalize our definitions and constructions to handle these cases, except for those of section 6 for confluent programs.

3 An operational semantics

The simpler semantics, the easier analysis, and here we propose a semantics which resembles the refined semantics but reintroduces nondetermiminism in certain steps, while retaining the essential properties concerning passive declarations.

We abstract away here, the textual order of rules, the order in which the state is searched for operational constraints when a call is looking for companions to form a head match, and the bookkeeping machinery that is needed to avoid loops by propagation rules applying to the same constraints over and over.

Any constraint in the state is classified as either *operational* or *unseen*, indicated by means of a respective subscript $^\bullet$ or $^\circ$. A state is a pair of a recursion stack being a finite sequence of constraints (operational and unseen mixed) and a constraint store which is a multiset of operational constraints.

We assume a fixed program given, and by an *application instance* of a rule $P \setminus Q$ `<=>` $G \mid R$ we mean an instance extended with operational and unseen markings as follows

$$(P^\bullet \ \setminus \ Q^\bullet \ \ \texttt{<=>} \ \ G \ \mid \ R^\circ)\sigma$$

where σ is any substitution with the property that it assigns a unique and unused variable to each variable of G and R which is not in P or Q, and so that $\models \exists\bar{x}(G\sigma)$ where $\bar{x}$ are the variables of $G\sigma$ not in $(P,Q)\sigma$. This definition is intended to simplify the semantics below in that head constraints of application instances may be identical to other operational constraints in the state (and thus CHR's notion of matching is handled in this way). Body constraints of application instances are marked as unseen with the intension that they stay as such until changed by an explicit step in the semantics. Notice that individual head constraints may be marked as passive as well; to simplify notation, passive markings are immaterial when constraints are compared by equality or subset operations. The symbols $\uplus$ refers to multiset union (with sequences viewed as multisets when relevant) and $\Subset$ to multiset inclusion.

A given query q is executed by a call $\mathsf{Run}(q^\bullet, \emptyset)$. This procedure, which is nondeterministic, is described as follows; ϵ denotes the empty stack, a dot is used as combined construction and concatenation operator. Letter A refers to a single constraint, and other letters to sequences or multisets as indicated by context.

1. **(halt)** $\mathsf{Run}(\epsilon, C^\bullet)$ halts for any $C^\bullet$.
2. **(simplify)** $\mathsf{Run}(A^\bullet \cdot q, C^\circ) \equiv$
   Select an application instance of a rule
   $P^\bullet \backslash Q_1^\bullet, A^\bullet, Q_2^\bullet \texttt{<=>} G | R^\circ$
   with $P^\bullet \uplus Q_1^\bullet \uplus Q_2^\bullet \Subset C^\bullet \uplus q$ and no passive mark on $A^\bullet$, and continue with
   $\mathsf{Run}(R^\circ \cdot q', C'^\bullet)$
   where q' and $C'^\bullet$ appear by removing each element of $Q_1^\bullet \uplus Q_2^\bullet$ exactly once from either q or $C^\bullet$.
3. **(propagate)** $\mathsf{Run}(A^\bullet \cdot q, C^\bullet) \equiv$
   Select an application instance of a rule
   $P_1^\bullet, A^\bullet, P_2^\bullet \backslash Q^\bullet \texttt{<=>} G | R^\circ$
   with $P_1^\bullet \uplus P_2^\bullet \uplus Q^\bullet \Subset C^\bullet \uplus q$ and no passive mark on $A^\bullet$, and continue with
   $\mathsf{Run}(R^\circ \cdot A^\bullet \cdot q', C'^\bullet)$
   where q' and $C'^\bullet$ appear by removing each element of $Q^\bullet$ exactly once from either q or $C^\bullet$.
4. **(activate)** $\mathsf{Run}(A^\circ \cdot q, C^\bullet) \equiv \mathsf{Run}(A^\bullet \cdot q, C^\bullet)$
5. **(shift)** $\mathsf{Run}(A^\bullet \cdot q, C^\bullet) \equiv \mathsf{Run}(q, C^\bullet \uplus \{A^\bullet\})$
6. **(solve)** $\mathsf{Run}((s = t)^\bullet \cdot q, C^\bullet) \equiv \mathsf{Run}((seq(R^\circ) \cdot q')\sigma, C'^\bullet\sigma)$
   where σ is an mgu of s, t and $R^\bullet$ the largest multiset of constraints affected[3] by σ so that $R^\bullet \Subset q \uplus C^\bullet$, and q' and $C'^\bullet$ appear by removing each element of $R^\bullet$ exactly once from either q or $C^\bullet$; $seq(R^\circ)$ is an arbitrary ordering of the elements of R° into a sequence.

Compared with [3], the choice of possible rule to apply is nondeterministic as well as when the shift step applies, e.g., for stopping a loop by the same propagation rule applying over and over again. The solve step is nondeterministic in the order in which the affected constraints are re-entered onto the recursion stack. Notice that the invention of application instances for steps 2 and 3 ensures that the complicated solve step only is involved in case the rule bodies or the initial query contains unifications.

4 Control-safety checked by local properties

Control-flow analysis and abstract interpretation for CHR programs are still a relatively sparsely investigated field, and another way to check control-safety is to check a slightly stronger property that can be checked locally for each step made in the operational semantics. However, this method needs the invention

[3] A substitution σ affects a variable x whenever $x\sigma \neq x$, and it affects any constraint containing a variable affected by σ.

of an invariant property for states, and it should also be proved that each step preserves the invariant.

Definition 2. *Let $\mathcal{P}(\Pi)$ be a program with passive declarations and* Inv *some property which may or may not hold for an execution state. A rule $\mathcal{P}(\rho) \in \mathcal{P}(\Pi)$ is* control-safe wrt Inv *whenever:*

- *For any state Σ satisfying* Inv*, $\mathcal{P}(\rho)$ can apply in Σ (step 2 or 3 of* Run*) if and only if ρ can apply, and such cases, the resulting successor state satisfies* Inv*.*

In case any rule $\mathcal{P}(\rho) \in \mathcal{P}(\Pi)$ is control-safe wrt Inv *and steps 4–6 (activation, shift, and solve) preserve* Inv*, we say that $\mathcal{P}(\Pi)$ is* locally control-safe *with respect to* Inv*.*

Control-safety for each rule can be checked by looking at that rule in isolation just assuming Inv and similarly for the other conditions of definition 2.

The following central property follows by a straightforward induction proof.

Proposition 1. *Let $\mathcal{P}(\Pi)$ be a program with passive declarations that are locally control-safe wrt* Inv*. Then the passive declarations of $\mathcal{P}(\Pi)$ are control-safe for the set of queries q for which $\langle q^\circ, \emptyset\rangle$ satisfies* Inv*.*

In other words, if we can design an invariant which accepts the desired class of queries and for which local safety can be shown for a progran $\mathcal{P}(\Pi)$, we have a guarantee that the indicated passive declarations do not change the semantics of Π, so we can let the CHR compiler optimize rule search based on those passive declarations.

5 Classes of programs with general strategies for control-safe passive declarations

5.1 CHR programs for bottom-up parsing

The CHR Grammar system (CHRG) [2] adds a grammar notation on top of CHR analogous to the way Definite Clause Grammars are added on top of Prolog. A CHRG is automatically compiled into a CHR program that works as a bottom-up parser and the system optionally adds passive declarations according to a predefined pattern. We take this as an example to illustrate how the method outlined in section 4 can be used to show the correctness of such a general strategy for a class of programs.

Example 2. The following CHR program implements a bottom-up parser for a small language; notice the passive declarations on all but rightmost head constraints.

```
np(N0,N1)#, v(N1,N2)#, np(N2,N3) ==> s(N0,N3).
token(N0,N1,peter) ==> np(N0,N1).
token(N0,N1,mary)  ==> np(N0,N1).
token(N0,N1,likes) ==>  v(N0,N1).
```

The parsing of a string can done by an initial query such as the following:

```
token(0,1,peter), token(1,2,likes), token(2,3,mary)
```

Let LRQ be the set of all queries of the form indicated by 2, consisting of `token` constraints, where the two first arguments recur as indicated and where the third argument can be whatever, so that it can be seen as a text string entered in left to right order.

We generalize the LRQ property into a state invariant by saying that a state $\langle q, C^\bullet\rangle$ satisfies Inv_{LR} whenever there exists a natural number $\mathbf{n}$ so that

- any constraint in the state is of form $c(n, m, \ldots)$ where n, m are natural numbers with $n < m$,
- any constraint in $C^\bullet$ and any operational constraint in q is of the form $c(n_0, n_1, \ldots)$ with $n_1 \leq \mathbf{n}$,
- Q is a sequence of the form $S \cdot U$ where S consists of operational constraints and U of unseen constraints; any constraint in S is of form $c(k, \mathbf{n}, \ldots)$; U is of the form $\langle \mathtt{token}(\mathbf{n}+1, \mathbf{n}+2, \ldots), \mathtt{token}(\mathbf{n}+2, \mathbf{n}+3, \ldots), \ldots\rangle$

With this, we can prove local control-safety for the program of example 2 from which follows control-safety.

In general, the CHRG system allows counterparts to propagation, simplification, and simpagation rules of CHR and allows also context-sensitive rules that may refer to symbols to the left and right of the string matched.

The proof of local control-safety for the sample program generalizes immediately to all CHR programs that results from the compilation of programs without right context, and with passive declarations applied to all but right-most constraints in each rule. See the detailed arguments in [2] which, however, does not use the framework of the present paper. In section 7 we report experiments measuring the actual speedup gained by passive declarations for such programs.

5.2 Programs with data structure and process constraints

Consider a class of CHR programs for which constraints can be separated into two disjoint classes D that we call *data structure constraints* and P called *process constraints*; let d and p be regular expression describing the respective classes of D and P constraints. Queries are restricted to the form d^*p^*. Each rule is of the form (commas implicit; notice that the order of constraints in heads is insignificant; for simplicity we show only propagation rules but the idea generalizes easily):

$$d^*p^+ \texttt{==>}\ p^*$$

When any D constraint is declared passive, the result is obviously control-safe.

In this case, the speedup is limited as it only affects the entering of the d^* part of the query into the constraint store. An example of the indicated pattern may be that d^* defines the edges of a graph and p^* are intended to check properties of that graph. We may generalize the pattern by introducing an intermediate class called F (decribed by regular expression f) of *formatting constraints* which, in the graph example, could evaluate the transitive closure of d^* corresponding to the paths in the graph. Rules are now of the forms

$$d^* f^+ \texttt{==>}\ f^* \qquad \text{and} \qquad d^* f^* p^+ \texttt{==>}\ p^*$$

In the first kind, d constraints can be made passive and in the second both d and f, and the result is also control-safe. Here the effective speedup should be expected to be significant (although our empirical tests below show something different). The program will execute in three phases, first D constraints are entered into the store with no rules checked, next the F constraints are derived but with no useless checks of applicability of rules referring to P; finally the P constraints execute with same efficiency independent of the passive declarations.

6 Confluent programs

Confluent CHR programs have the property that the final state is independent of the order in which rules are applied. Precise definitions and ways of showing confluence of programs are given by [1]. In other words, any possible derivation for a confluent program is, so to speak, equally good, and possible derivations can be disregarded as long as one derivation remains. ¿From this argument, it follows that control-safety is an unnecessarily strong restriction for achieving answer-safety for confluent programs.

There is, in fact, a straightforward criterion that ensures answer-safety for programs proved by other means to be confluent.

Definition 3. *Let $\mathcal{P}$ be a set of passive declarations of a program $\mathcal{P}(\Pi)$ where Π is confluent. We say that $\mathcal{P}$ is* non-blocking *if, in any state in which a rule of Π can apply, there is some rule of $\mathcal{P}(\Pi)$ that can apply.*

This condition is obviously sufficient for answer-safety, and it is also necessary. To see this, assume that the passive declarations of a program $\mathcal{P}(\Pi)$ are not non-blocking, i.e., there is a combination of constraints Q for which no rule of $\mathcal{P}(\Pi)$ can apply, but there is one in Π that can apply. Obviously, with Q as query, the final state reached by $\mathcal{P}(\Pi)$ is different from the one reached by Π.[4]

It should be kept in mind, however, that non-blocking passive declarations added to a confluent program may not be an optimization, but may in worst cases slow down execution considerably. This will be the case whenever the actual implementation of a confluent program Π chooses a short derivation for some query but $\mathcal{P}(\Pi)$ bypasses this option and goes a long and laborious way to find the same result. The opposite may, of course, also happen. There is a simple criterion for testing non-blocking, one rule at a time.

Proposition 2. *Let $\mathcal{P}$ be a set of passive declarations of a program $\mathcal{P}(\Pi)$ where Π is confluent. Then $\mathcal{P}$ is non-blocking whenever for any rule $\mathcal{P}(\rho) \in \mathcal{P}(\Pi)$ and*

[4] In this argument, we have assumed the absence of rules that do not affect the state such as `a,b==>true`.

any passive constraint c_ρ in its head, there is some rule $\mathcal{P}(\pi) \in \mathcal{P}(\Pi)$ for which the following hold; h_ρ and h_π refer to the heads of $\mathcal{P}(\rho)$ and $\mathcal{P}(\pi)$, and G_ρ and G_π to their guards:

- *The predicate of c_ρ occurs non-passively in the head of π as c_π.*
- *It holds that $\exists(c_\rho \wedge G_\rho) \rightarrow \exists(c_\pi \wedge G_\pi)$ and $\exists(h_\rho \wedge G_\rho) \rightarrow \exists(h_\pi \wedge G_\pi)$.*

The separate test for c_ρ, c_π in the last condition is only necessary in case π has another and passive occurrence of the predicate of c_ρ in its head. Notice that all tests can be made by a syntactic subsumption check combined with a constraint solver for the built-in predicates applied in the guards.

Example 3. Assume the following two rules are part of a confluent program that has been extended with passive declarations.

```
a(X), b(7)# ==> p(X).
b(X) <=> X =< 10 | q(X).
```

The passive declaration is non-blocking in the sense of the proposition since we have $\exists(\texttt{a(X)} \wedge \texttt{b(7)}) \rightarrow \exists(\texttt{b(X)} \wedge \texttt{X=<10})$. (Proper definitions of predicates `p` and `q` are expected for this program to be confluent.)

Example 4. In a rule with symmetry in the head such as

```
p(X), p(Y) ==> p(X,Y).
```

any of the two head constraints can be made passive (but not both), and we will have the non-blocking condition satisfied with π being the same rule as ρ. In fact, such an optimization is made already in existing CHR compilers, cf. [7].

The opposite of proposition 2 is not the case in general.

Example 5. Consider the following program.

```
p(1) <=> q(1).    p(X) <=> X\=1 | q(X).    p(X)# <=> q(X).
```

Its passive declaration is non-blocking but it does not satisfy the conditions of proposition 2. It seems, however, straightforward to generalize the proposition to recognize such cases as well.

The notions of confluence and non-blocking passive declarations can be refined when a query restriction and state invariant are given as to allow more passive declarations. The local test of proposition 2 must, then, be extended as to ensure that each step made by the operational semantics preserves the invariant; the construction seems straightforward and analogous to the one in section 4.

7 An estimation of the speedup gained by passive declarations

In order to give a mathematical characterization of the speedup gained, a very detailed operational model would be needed, far more complex than the operational semantics we use and the refined semantics that we have referred to.

Informally, we can point out the following factors for control-safe passive declarations. To observe the largest effect, assume that a large proportion of all head constraints have been made passive.

- When a constraint is called, the number of head constraints that it matches in the entire program may fall with a factor proportional with the program size (e.g., down to one).
- The work that would have been spent for finding out that each one of the unsuccessful matches did not lead to a full match may be exponential: Any combination of constraints in the store may need to be tried out.

So the conclusion is that adding passive declarations in a control-safe way may result in speedups from zero to exponential. For the answer-safe and not control-safe cases, the speedup can be arbitrarily large as any "sick" branch in the tree of possible computation paths may be pruned (but, as noted above, the speedup may also be negative in some cases).

We made some test runs for grammars in CHR in SICStus Prolog with and without passive declarations on all but the rightmost constraint in each head as described in section 5. All test were run on a Macintosh G4 1.33GHz running OS X, Version 10.3.9 and SICStus Prolog 3.12.0 and compiled with Prolog's optimizing compiler.

We used a series of highly ambiguous grammars that produce an exponential number of nodes measured in the length of the input string. We tested with 2–5 constraints in the head of each rules; the one for 3 constraints includes (with passive declarations added) these rules:

```
token(N0,N1,a) ==> a(N0,N1).
a(N0,N1)#, a(N1,N2)#, a(N2,N3) ==> a(N1,N3).
```

In addition it has 10 identical rules of the form

```
a(N0,N1)#,  a(N1,N2)#, a(N2,N3) ==> dummy.
```

We tested different input strings of length up 11 and longest running time of 45 sec's, which resulted in speedup factors of 1.5–2 with no clear correlation to the length of the input string or the number of tokens in the store.

To test the model of section 5.2, we used a program that first finds all shortest paths in a graph, and then derives some strange properties from edge and paths constraints. Surprisingly, we could observe speedups of only 0–7% for different runs. Trying to explain this, we observe that when, say, an edge constraint is called, the checking suppressed concerns exclusively rules where the constraint store at the given moment contains no constraints at all for the other predicates mentioned in the heads of those rules. We will expect that the SICStus compiler produces code that finds out very quickly if head predicates are empty.

Finally, we did a test constructed so that the advantages of passive declarations should be obvious and where the compiler should have no chance to optimize as the gain depends heavily on the class of queries. The passive declarations in the following one-line program

```
a(X)#, b(Y)#, c(Z) ==> S is X+Y+Z, S<5 | d(X,Y,Z)
```

are control-safe for the class of queries of the form `c(1)`, ..., `c(5)`, `a(1)`, ..., `a(`n`)`, `b(1)`, ..., `b(`n`)`, `c(1)`, ..., `c(5)`. Here we observed a speedup factor proportional to n: around 10 for $n = 50$, 20 for $n = 100$, and 40 for $n = 200$, with the running times for the last example being 76 sec. versus less than 2 sec. without and with passive declarations.

8 Related work

The general outline of the compiler architecture for CHR is explained in [8], where CHR is compiled into Prolog using attributed variables. This compiler embeds the so-called refined semantics later formalized by [3], and which is necessary for an understanding of passive declarations; however, [3] do not consider passive declarations as we do with our operational semantics. As mentioned, our semantics is considerably simpler as it is intended mainly for manual proofs but it also ignores some aspects such as the order in which clauses are tried out. Adding a treatment of passive declarations to [3]'s semantics, which will be fairly straightforward, will obviously make it possible to explain correctness of more kinds of declarations in a systematic way.

In [6], the authors describe a number of optimizations made in the CHR compiler of the HAL system, most of which do not refer to passive declarations or can be explained as such. However, they explain so-called continuation optimizations where, depending on success of failure of a called constraint to trigger a rule (by gathering companion constraints to match the full head plus satisfying the guard), it is determined whether the next occurrence of that constraint predicate in a rule head needs to be checked.

Abstract interpretation is a very powerful technique for program analysis and a wide range of methods and tools are available for other types of programming languages, but as we have mentioned, CHR has a more complicated control structure than most other languages, so existing methods cannot be directly taken over. In the lack of such tools, we have made some experiments by analyzing a meta-interpreter for CHR, written in Prolog as a straightforward replication of our operational semantics. We then analyzed its behaviour for given CHR programs using a tool for abstract interpretation of Prolog [5], but it was not possible to obtain sufficiently precise information about CHR's execution state as to make interesting predictions concerning the control flow.

Very recently, [11] has proposed a framework for abstract interpretation of CHR, which we have not yet had the possibility to apply for our purpose, but this is an obvious next step.

Optimizations and analyses made by a compiler for CHR cannot take into account restrictions to classes of relevant queries as we have done, but we may refer to recent work [13, 10] that also employs such restrictions.

The present paper concerns optimizations of CHR programs which can be done at the source language level by means of passive declarations. In contrast to this, or should we say in competition with, the available CHR compilers are under continual improvement with optimizations done at lower levels which may be unpublished or difficult to trace. As we have noticed, in many cases the addition of passive declarations gives less speedup that expected which indicates that CHR compilers already do many similar optimizations, many of which go for very specific cases. We may mention a few such optimizations that we are aware of. HAL and SWI-Prolog's CHR compilers employs a "never stored" optimization [9, 6]: In case of a simplification rule of form `p(X1, ..., X`$_n$`) <=> true |` $\cdots$, passive declarations can be added to other rules containing `p` in the head; all constraints other than `p` in such rules can be treated as passive. This is an example of a control-safe optimization but which is obviously not caught by the local control-safety criterion we have proposed. In addition, [9] reports a method adding possible passive declarations to simplification rules of the form `p(` $\cdots$ `) \ p(` $\cdots$ `) <=> true |` $\cdots$, when a condition of functional dependency among the arguments is met. A related optimization based on so-called occurrence subsumption (also for simplification rules) is described by [12].

9 Conclusion and perspectives

We have given a framework for reasoning about passive declarations of CHR programs and indicated ways of showing their correctness in the sense that they do not change the final results. We formalized this as a notion of answer-safety, and indicated a stronger and in many cases easier to check property called control-safety.

We indicated different methods to check or identify control-safe patterns of passive declarations, and we showed how our methods can be applied to prove once and for all the correctness of strategies for adding passive declarations to particular classes of CHR programs. For programs proved (by other means) to be confluent, we have proposed a simple and easily mechanizable method for adding answer-safe passive declarations that are not necessarily control-safe.

Finally we notice that our analysis and optimization were made entirely at the high level of the source language (CHR) and a relatively abstract operational semantics. Moving our ideas down to a more detailed level of an actual compiler (cf. [6]) gives more freedom for optimizations, but we will also lose transparency in the presentation.

Acknowledgment

This work is supported by the CONTROL project, http://control.ruc.dk, funded by the Danish Natural Science Research Council. The author wants to thank

John Gallagher for helpful discussions and for cooperation on various experiments with abstract interpretation. Also thanks to anonymous reviewers of an earlier version of this paper for additional references and insightful comments on CHR compilers.

References

1. Slim Abdennadher, Thom W. Frühwirth, and Holger Meuss. On confluence of Constraint Handling Rules. In Eugene C. Freuder, editor, *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*, volume 1118 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.
2. Henning Christiansen. CHR Grammars. *Theory and Practice of Logic Programming, Special Issue on Constraint Handling Rules*, 2005. To appear.
3. Gregory J. Duck, Peter J. Stuckey, Maria J. García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2004.
4. Thom W. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programmming*, 37(1-3):95–138, 1998.
5. John P. Gallagher and Kim S. Henriksen. Abstract domains based on regular types. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 27–42. Springer, 2004.
6. Christian Holzbaur, Maria 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*, 2005. To appear.
7. Christian Holzbaur, Maria J. García de la Banda, David Jeffery, and Peter J. Stuckey. Optimizing compilation of Constraint Handling Rules. In Philippe Codognet, editor, *Logic Programming, 17th International Conference, ICLP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2237 of *Lecture Notes in Computer Science*, pages 74–89. Springer, 2001.
8. Christian Holzbaur and Thom W. Frühwirth. Compiling Constraint Handling Rules into Prolog with attributed variables. In Gopalan Nadathur, editor, *Principles and Practice of Declarative Programming, International Conference PPDP'99, Paris, France, September 29 - October 1, 1999, Proceedings*, volume 1702 of *Lecture Notes in Computer Science*, pages 117–133. Springer, 1999.
9. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: implementation and application. In *Proceedings of the First Workshop on Constraint Handling Rules: Selected Contributions, Ulm, Germany, May 2004*, volume 2004-01 of *Ulmer Informatik-Berichte*. Universität Ulm, Germany, 2004.
10. Tom Schrijvers and Thom W. Frühwirth. Analysing the CHR implementation of Union-Find. In Armin Wolf, Thom W. Frühwirth, and Marc Meister, editors, *W(C)LP, 19th Workshop on (Constraint) Logic Programming, Ulm, Germany, February 21-23, 2005*, volume 2005-01 of *Ulmer Informatik-Berichte*. Universität Ulm, Germany, 2005.

11. Tom Schrijvers, Peter J. Stuckey, and Gregory J. Duck. Abstract interpretation for Constraint Handling Rules. In Pedro Barahona and Amy P. Felty, editors, *Proceedings of the 7th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 11-13 2005, Lisbon, Portugal*, pages 218–229. ACM, 2005.
12. Jon Sneyers, Tom Schrijvers, and Bart Demoen. *Guard and Continuation Optimization for Occurrence Representations of CHR.* Technical report, CW420, K.U.Leuven, Department of Computer Science, 2005.
13. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Guard simplification in CHR programs. In Armin Wolf, Thom W. Frühwirth, and Marc Meister, editors, *W(C)LP, 19th Workshop on (Constraint) Logic Programming, Ulm, Germany, February 21-23, 2005*, volume 2005-01 of *Ulmer Informatik-Berichte*, pages 123–134. Universität Ulm, Germany, 2005.
14. Swedish Institute of Computer Science. *SICStus Prolog user's manual, Version 3.12*, 2005. Available at `http://www.sics.se/isl`