Aggregates in CHR

Jon Sneyers
Peter Van Weert
Tom Schrijvers
Bart Demoen

Report CW 481, March 2007



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Aggregates in CHR

Jon Sneyers
Peter Van Weert
Tom Schrijvers
Bart Demoen

Report CW 481, March 2007

Department of Computer Science, K.U.Leuven

Abstract

We propose an extension of the Constraint Handling Rules language with aggregates like `sum`, `count`, `findall`, and `min` in the heads of rules. We define the semantics of aggregate expressions formally and informally. Our prototype implementation allows nested aggregate expressions over guarded conjunctions of constraints, using either an on-demand or an incremental computation strategy. Case studies demonstrate that by using aggregates, the program size can be reduced significantly, with an acceptable constant run-time overhead.

Keywords : Constraint Handling Rules, aggregates, language design, semantics, source-to-source transformation.
CR Subject Classification : D.3.2 [Programming Languages] Language Classifications — Constraint and logic languages.

Aggregates in Constraint Handling Rules

Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium
FirstName.LastName@cs.kuleuven.be

Abstract. We propose an extension of the Constraint Handling Rules language with aggregates like `sum`, `count`, `findall`, and `min` in the heads of rules. We define the semantics of aggregate expressions formally and informally. Our prototype implementation allows nested aggregate expressions over guarded conjunctions of constraints, using either an on-demand or an incremental computation strategy. Case studies demonstrate that by using aggregates, the program size can be reduced significantly, with an acceptable constant run-time overhead.

1 Introduction

Constraint Handling Rules (CHR) [5] is a general-purpose programming language extension, based on multi-headed, guarded multiset rewrite rules. Traditional applications of CHR lay in the area of constraint solving, but CHR is also used extensively in a wide range of other applications, such as type system design, natural language processing and multi-agent systems.

CHR aims at supporting a very high-level, declarative programming style. While imperative programs explicitly specify an algorithm to achieve some goal, declarative programs simply describe the goal, leaving implementation details to the underlying language. This considerably shortens development time, and vastly improves a program's understandability, maintainability and robustness.

In practice, however, there are programming idioms where CHR's conciseness and expressiveness is lacking. This paper identifies and addresses one frequently recurring instance, namely the aggregation of information from nontrivial, *possibly unbounded* parts of the constraint store. However, each individual CHR rule only considers a fixed, *bounded* number of constraints, equal to the number of conjuncts in its head. Aggregations therefore always require some explicit encoding, using multiple auxiliary rules and constraints. Such ad hoc approaches are repetitive, cumbersome and error-prone, and the resulting auxiliary constructs often cross-cut the entire program. In other words, they elevate all advantages of declarative programming.

With language support for aggregates the programmer can express the aggregate logic concisely in a single self-contained rule, exhibiting all advantages of declarative programming. Aggregates are already an established feature of several related declarative languages. Some, most notably SQL [2], only offer a fixed set of predefined aggregate functions. In practice however, information often has to be aggregated in application-specific ways. Therefore, we propose an extensible aggregate framework that caters for user-defined aggregate functions.

Our prototype implementation rewrites CHR programs with aggregates into CHR programs without. By default, aggregate computation is on-demand, but our infrastructure provides high-level control over performance through optional incremental aggregate computation. Evaluation of the resulting code indicates only a modest overhead compared to manual approaches.

Overview The rest of this paper is structured as follows. In Section 2 we present an overview of our new aggregates language construct. Next, Section 3 provides a formal definition of aggregates in CHR. Section 4 discusses our prototype implementation which is evaluated in Section 5. Finally, Section 6 reviews related work and Section 7 concludes.

2 Aggregates

This section introduces an extension of CHR with aggregates, in which heads of CHR rules may contain aggregates ranging over CHR constraints.

2.1 Motivating Example

As pure CHR is already Turing complete [17], does it need another language feature? Aggregates certainly do not add to the computational power of CHR. Nevertheless, we will show that they are invaluable when it comes to expressivity, maintainability and conciseness.

Suppose that the constraints `account(AccountId,ClientId,Type,Balance)` and `client(ClientId)` constitute a (simplified) representation of the accounts and the clients of a bank. One of the business rules of the bank states: "*A* platinum *client is a client whose account balance is $25,000 or more*", or, in CHR:

```
client(C), account(_,C,_,B) ==> B >= 25000 | platinum(C).
```

However, as clients are allowed to have multiple accounts, the bank later prefers the business rule: "*A platinum client is a client whose* accumulated *sum of account balances is $25,000 or more*". We compare three different approaches for expressing this extended rule: the first two are possible in current CHR systems, whereas the third one uses aggregates.

Naive approach. If the maximum number of accounts per client is limited to some fixed number n, all possible cases are expressed in CHR as:

```
client(C), account(_,C,_,B) ==> B >= 25000 | platinum(C).
...
client(C), account(_,C,_,B1), ..., account(_,C,_,Bn)
    ==> B1+...+Bn >= 25000 | platinum(C).
```

Software engineering methodology dictates that the above replication of code is highly undesirable: it is hard to read and hard to maintain. This approach also scale very badly performance-wise, as the number of combinations tried during matching increases exponentially. Moreover, exhaustively enumerating all possible cases is impossible if n is unbounded.

Common approach. A more concise solution, commonly used by CHR practitioners, is to introduce an auxiliary constraint `accumulated_balance/2`:

```
client(C), accumulated_balance(C,Sum) ==> Sum > 25000 | platinum(C).
```

This concisely captures the logic of platinum clients in a single rule. A second advantage over the naive approach is that it facilitates an unbounded number of accounts per client. This approach remains, nevertheless, inadequate, because it necessitates the maintenance of the accumulated balance. This inherently is a cross-cutting concern, as it requires invasive modifications to all parts of the original code that alter the balance of an account:

```
deposit(A,X), account(A,C,T,B) <=> account(A,C,T,B+X).
...
withdraw(A,X), account(A,C,T,B) <=> B > X, account(A,C,T,B-X).
```

All these rules, spread throughout the entire program, have to be adjusted to update `accumulated_balance` accordingly:

```
deposit(A,X), account(A,C,T,B), accumulated_balance(C,Acc) <=>
        account(A,C,T,B+X), accumulated_balance(C,Acc+X).
...
withdraw(A,X), account(A,C,T,B), accumulated_balance(C,Acc) <=>
  B > X, account(A,C,T,B-X), accumulated_balance(C,Acc-X).
```

Also, the accumulated balance has to be initialized for new clients:

```
client(C) ==> accumulated_balance(C,0).
```

Many variations to the above maintenance scheme can be concocted, but they all require similar modifications scattered throughout the entire program. As a result, this approach also displays poor compliance with common software quality criteria: it is very error-prone, and it impairs the readability and maintainability of the program, as the logic of many rules becomes tangled with obfuscating auxiliary code.

Aggregates. The use of aggregates shares the benefits of the previous approach, whilst dispensing with its drawbacks. Using an aggregate expression (in italics), the platinum client business rule is again declaratively expressed in a single rule, independent of the number of accounts:

```
client(C), sum(B,account(C,_,B),Sum) ==> Sum > 25000 | platinum(C).
```

No further changes to the program are required. A perfectly correct behavior is already guaranteed implicitly by the aggregate's semantics.

As a result, the program is more declarative, readable and maintainable. The programmer's productivity is improved, because he is relieved from the cumbersome and repetitive task of implementing aggregates, and can entirely focus on his application domain.

2.2 Syntax and Informal Semantics

Next to a collection of predefined aggregates, our framework provides a general mechanism for the declaration of user-defined, application-tailored aggregates. We now present the proposed syntax for both aggregate types, followed by an informal description of their operational semantics. A formal definition of aggregate semantics is given in Section 3.3.

Predefined aggregates. Table 1 lists the predefined aggregates in our system.

Aggregate	*Meaning*	*Synonyms*
`nb(`G`,`C`)`	G matches C times	`count`
`collect(`X`,`G`,`C`)`	C is a list of X's for every match of G	`findall`
`exists(`G`)`	G exists (`nb(`G`,`C`),` $C > 0$)	
`\+` G	G does not exist (`nb(`G`,0)`)	`no, none`
`forall(`G`,`C`)`	for every match of G, condition C holds	`implies`
`min(`X`,`G`,`M`)`	M is the minimum of X over all matches of G	`minimum`
`argmin(`X`,`G`)`	G is matched such that X is minimal	`findmin`
`takemin(`X`,`G`)`	like `argmin(`X`,`G`)`, but G is also removed (analogously for `max`/3, `argmax`/2, `takemax`/2)	`rmin`
`sum(`X`,`G`,`R`)`	R is the sum of X over all matches of G	
`prod(`X`,`G`,`R`)`	... product ...	`product`
`avg(`X`,`G`,`R`)`	... (arithmetic) average ...	`average`
`stddev(`X`,`G`,`R`)`	... standard deviation ...	
`var(`X`,`G`,`R`)`	... variance ...	`variance`

Table 1. Predefined aggregates. The goal, G, is an arbitrary conjunction of CHR constraints, guards and aggregates. For all arithmetic aggregates, X has to be a ground arithmetic expression at runtime. For `collect`/3, X is just an arbitrary template, as e.g. in the well-known `findall`/3 ISO Prolog predicate [1].

User-defined aggregates. Often information has to be aggregated in application-specific ways. Therefore, we designed a general high-level mechanism that enables CHR end-users to create their own *user-defined aggregates*:

```
aggregate(Init, Inc, Dec, Final, Element, Goal, Result)
```

The meaning of the arguments is as follows:

`Init`	returns the initial working value;
`Inc`	takes a working value and an element and returns a new (incremented) working value;
`Dec`	takes a value and an element and returns a decremented value;
`Final`	takes a working value and returns the result;
`Element`	is a template to describe an element for a given `Goal`;
`Goal`	is a conjunction of CHR constraints, guards, and aggregates;
`Result`	returns the result of the aggregate.

The first four arguments have to be manifest, and declare how the aggregate has to be computed. Their concrete format is host-language dependent: in CHR(C) for example function pointers could be used, and in CHR(Java) e.g. methods of the working value object. This paper considers Prolog as the host-language, where the first four are terms representing the Prolog predicates that have to be called (after extending them with some extra arguments). The last three arguments are analogous to those in Table 1. The concrete operational semantics of the arguments is explained below.

The general `aggregate/7` construct is expressive enough to formulate any aggregate function. In fact, all predefined aggregates are implemented through it: `sum(X,G,S)`, e.g., is equivalent to `aggregate(=(0),plus,minus,=,X,G,S)`. More detailed information on the different types of aggregates that can be expressed is provided in Section 4.4.

Informal semantics. Using `sum(X,G,S)` (cf. supra) as a running example, we begin with explaining how the result of an aggregate is computed.

First, the working value W_0 of the aggregate is initialized by calling `Init`(W_0). In the case of `sum`, this results in an initial working value $W_0 = 0$. This value is then incremented once for each of the n matchings of `Goal`, by calling `Inc`(W_{i-1}, `Element`, W_i) for $1 \leq i \leq n$. For `sum`, the increment predicate `plus` adds the working value and the value of `Element` to get a new working value. Finally, the last working value W_n is finalized by calling `Final`(W_n, `Result`). Often, like for `sum`, this finalizer predicate simply unifies the last working value with `Result`. The aggregate's computation, and in particular the finalizer predicate, is allowed to fail. In that case, the aggregate is undefined and the rule containing it is not applicable. For instance, `max` and `avg` are undefined when there are no elements. The decrement predicate `Dec` is discussed later, in Section 4.4, which describes an alternative computation strategy.

Operationally, a rule containing an aggregate is tried when one of the other head constraints is inserted or triggered (we call this a *passive* aggregate computation). It is also tried when one of the CHR constraints in the `Goal` of the aggregate is inserted, triggered, or removed (this is an *active* aggregate computation). In the banking example of Section 2.1, there is a passive aggregate computation when a new `client` is added, and an active aggregate computation when an `account` is added, removed, or updated.

Syntactic shortcuts. Because the general `aggregate/7` notation is overly verbose, it is quite useful to use a macro facility for defining abbreviations. The CHR host language usually offers such a facility, e.g. term expansion in Prolog, or C's macro language. We prefer to integrate some macro facility in the CHR language itself for reasons of portability and possible scheduling conflicts with CHR compilation[1]. The syntax is as follows:

`:- chr_expansion` *head* `--->` *body*`.`

[1] In Prolog, CHR compilation itself is often realized through term expansion.

This replaces any occurrence of *head* in the head of a rule with *body*, where *head* is a single atom, and *body* can be any conjunction of atoms. Predefined aggregates behave as if defined this way, e.g.:

```
:- chr_expansion sum(E,G,R) ---> aggregate(=(0),plus,minus,=,E,G,R).
```

This allows to name new aggregates and even to give application-specific names to (special cases) of existing aggregates. For example:

```
:- chr_expansion in_degree(N,C) ---> count(edge(_,N), C).
:- chr_expansion out_degree(N,C) ---> count(edge(N,_), C).
```

Not only are these user-defined aggregate names more user-friendly then the general `aggregate`/7 construct, they also vastly increase the readability and maintainability of the resulting programs.

The above is just a rather primitive, ad-hoc macro facility. More advanced source-to-source transformations are possible through our *meta CHR rules* powered preprocessor, briefly introduced in Section 4.1. However, meta CHR rules can, in the current implementation, not be written directly in the user program. The advantage of the macro expansion aid is that it can be used directly in the user program. Once the restriction that meta rules have to be defined in a separate file is lifted, the macro expansion facility will become obsolete.

Complex aggregate goals and nested aggregates. Until now we have only shown examples of aggregates over a simple `Goal`, i.e., consisting of one CHR constraint. Of course, more complex aggregate goals can also be used. For example, `count((platinum(C), account(_,C,_,_)), N)` counts the number of accounts owned by platinum clients. Its goal is a conjunction of two constraints.

Even more expressivity is realized by allowing *nested aggregates*, that is, aggregate expressions inside the goal of another aggregate. For example, to get the client `C` with the largest total balance, we can use the following nested aggregate expression: `argmax(S, (client(C), sum(B,account(_,C,_,B),S)))`.

3 Formal Aggregates Definition

In order to define aggregates formally, a short introduction to CHR is necessary. In this section we briefly recapture the syntax and (operational) semantics of CHR. More information can be found in [5] and [14]. We then extend the semantics to allow `aggregate`/7 expressions. Finally, we give a brief introduction to the refined operational semantics [4].

3.1 Syntax of CHR

CHR is embedded in a host language that provides data types and a number of predefined constraints. These constraints are called host language constraints or *built-in constraints*. The host language considered in this paper is Prolog.

Its host language constraints are unification, Prolog built-ins and other (user-defined) Prolog predicates. Its data types are Prolog variables and terms.

CHR constraint symbols are drawn from the set of predicate symbols, denoted by a functor/arity pair. *CHR constraints* are atoms constructed from these symbols and the data types offered by the host language.

A CHR program $\mathcal{P}$ consists of a sequence of CHR rules, of the form

$$\text{name} \ @ \ \ H_k \setminus H_r \iff g \mid B$$

where H_k and H_r are conjunctions of CHR constraints called the (kept and removed) *heads*, g is a conjunction of built-in constraints called the *guard*, and B is a conjunction of (CHR and built-in) constraints called the *body*.

The name is optional and unique; rules without a name get a unique implicit name. The guard "$g \mid$ " is optional; if omitted, it is considered to be "$true \mid$ " (the empty conjunction). If H_k is empty the rule is a *simplification* rule. If H_r is empty the rule is a *propagation* rule and the symbol " $\Longrightarrow$ " is used instead of " $\Longleftrightarrow$ ". If both parts are non-empty, the rule is a *simpagation* rule. At least one of H_r and H_k must be non-empty.

Logically, a simplification rule corresponds to an equivalence : $g \rightarrow (H_r \leftrightarrow B)$, while a propagation rule corresponds to an implication : $g \rightarrow (H_k \rightarrow B)$.

Intuitively, the operational semantics of a CHR rule is the following: if all heads are in the constraint store and the guard is satisfied, then the removed part of the head (H_r) is deleted from the store and the body is executed.

3.2 The Operational Semantics ω_t

Formally, the execution of a CHR program follows the operational semantics ω_t, sometimes also called *theoretical* or *high-level* operational semantics. The ω_t semantics is formulated as a state transition system. Transition rules define the relation between an execution state and its subsequent execution state.

Definition 1. *An* execution state σ *is a tuple* $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$. *The* goal $\mathbb{G}$ *is a multiset of constraints to be rewritten to solved form. The CHR constraint* store $\mathbb{S}$ *is a set of* identified *CHR constraints that can be matched with rules in the program* $\mathcal{P}$. *An* identified *CHR constraint* $c\#i$ *is a CHR constraint* c *associated with some unique integer* i, *the* constraint identifier. *This number serves to differentiate among copies of the same constraint. We introduce the functions* $chr(c\#i) = c$ *and* $id(c\#i) = i$, *and extend them to sequences and sets of identified CHR constraints in the obvious manner, e.g.* $id(S) = \{c | c\#i \in S\}$. *Note that* $chr(\mathbb{S})$ *is a multiset even though* $\mathbb{S}$ *is a set. The* built-in constraint store $\mathbb{B}$ *is the conjunction of all built-in constraints that have been passed to the underlying solver. The* propagation history $\mathbb{T}$ *is a set of tuples, each recording the identities of the CHR constraints that fired a rule, and the name of the rule itself. This is necessary to prevent trivial non-termination for propagation rules. Finally, the counter* $n \in \mathbb{N}$ *represents the next integer to identify a CHR constraint.*

Definition 2. *The set of* matching substitutions *is defined as follows, where H are CHR constraints and G are built-in constraints:*

$$\mathsf{matchings}(H \wedge G, S_h, \mathbb{S}, \mathbb{B}) = \left\{\theta \;\middle|\; chr(H) = \theta(S_h) \wedge \mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}}(\theta \wedge G)\right\}$$

Given a CHR program $\mathcal{P}$, transitions are defined by the binary relation $\rightarrowtail_{\mathcal{P}}$ shown in Figure 1. Execution proceeds by exhaustively applying the transition rules, starting from an initial state of the form $\langle \mathbb{G}, \emptyset, \mathit{true}, \emptyset \rangle_1$.

1. Solve.	$\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle \mathbb{G}, \mathbb{S}, c \wedge \mathbb{B}, \mathbb{T} \rangle_n$ where c is a built-in constraint and $\mathcal{D}_{\mathcal{H}} \models \bar{\exists}_{\emptyset}\mathbb{B}$.
2. Introduce.	$\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle \mathbb{G}, \{c\#n\} \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$ where c is a CHR constraint and $\mathcal{D}_{\mathcal{H}} \models \bar{\exists}_{\emptyset}\mathbb{B}$.
3. Apply.	$\langle \mathbb{G}, H_1 \cup H_2 \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle C \uplus \mathbb{G}, H_1 \cup \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$ where $\mathcal{D}_{\mathcal{H}} \models \bar{\exists}_{\emptyset}\mathbb{B}$ and $\mathcal{P}$ contains a rule $r \;@\; H_1' \backslash H_2' \iff g \mid C$ and $\theta \in \mathsf{matchings}((H_1', H_2', g), H_1 \cup H_2, \mathbb{S}, \mathbb{B})$ and $h = (r, id(H_1, H_2)) \notin T$

Fig. 1. The transition rules of the theoretical operational semantics ω_t, defining $\rightarrowtail_{\mathcal{P}}$.

3.3 Adding Aggregates to ω_t

We modify the definition of ω_t-transitions to deal with nested `aggregate`-expressions as introduced in Section 2.2. We use two mutually recursive definitions:

Definition 3. *We redefine the set of matching substitutions as follows:*

$$\mathsf{matchings}'(A \wedge H \wedge G, S_h, \mathbb{S}, \mathbb{B})$$

$$= \left\{\theta \;\middle|\; chr(H) = \theta(S_h) \wedge \mathcal{D}_{\mathcal{H}} \models \mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}}(\theta \wedge G \wedge \mathsf{agg_cond}(A, S_h \cup \mathbb{S}, \mathbb{B}))\right\}$$

where A is a conjunction of aggregates, H is a conjunction of CHR constraints, and G is a conjunction of build-in constraints.

Definition 4. *We define the* aggregate condition *as follows, for an aggregate A of the form* $\mathtt{aggregate}(s, i, f, d, X, G, R)$*, a CHR store $\mathbb{S}$ and a built-in store $\mathbb{B}$:*

$$\mathsf{agg_cond}(A, \mathbb{S}, \mathbb{B}) = \Big[\; s(V_0) \wedge \Big(\{\theta_1, \ldots, \theta_n\} = \bigcup_{H \subseteq \mathbb{S}} \mathsf{matchings}'(G, H, \mathbb{S}, \mathbb{B})\Big)$$

$$\wedge \bigwedge_{k=1}^{n} i(V_{k-1}, \theta_k(X), V_k) \;\wedge\; f(V_n, R)\Big]$$

where $V_0, \ldots, V_n$ are fresh variables. We extend this condition to (empty) conjunctions of aggregates in the obvious way:

$$\mathsf{agg_cond}(A \wedge B, \mathbb{S}, \mathbb{B}) = \mathsf{agg_cond}(A, \mathbb{S}, \mathbb{B}) \wedge \mathsf{agg_cond}(B, \mathbb{S}, \mathbb{B})$$
$$\mathsf{agg_cond}(\mathit{true}, \mathbb{S}, \mathbb{B}) = \mathit{true}$$

The definition is unambiguous if the increment predicate i corresponds to a commutative operator. Finally, we slightly modify the **Apply** transition to use the modified set of matching substitutions $\mathsf{matchings}'$:

3. Apply'. $\langle \mathbb{G}, H_1 \cup H_2 \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{\mathcal{P}} \langle C \uplus \mathbb{G}, H_1 \cup \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$
where $\mathcal{D}_{\mathcal{H}} \models \bar{\exists}_{\emptyset} \mathbb{B}$ and $\mathcal{P}$ contains a rule $r \ @ \ A, H_1' \backslash H_2' \iff g \mid C$ and $\theta \in \mathsf{matchings}'((A, H_1', H_2', g), H_1 \cup H_2, \mathbb{S}, \mathbb{B})$ and $h = (r, id(H_1, H_2)) \notin T$

3.4 The Refined Operational Semantics ω_r

For a proper understanding of the implementation section, some basic knowledge of the *refined operation semantics* ω_r is required. More detailed information can be found in [4].

The ω_r semantics formalize the execution mechanism of most current CHR systems, including the one underlying our prototype implementation, the K.U.Leuven CHR system [14, 15]. The theoretical semantics ω_t, presented above in Section 3.2, is highly non-deterministic, and relies on writing confluent programs to have a meaningful behavior. However, many, if not most, existing CHR programs are nonconfluent under ω_t. Instead, they rely on known properties of the execution order of the underlying CHR system. ω_r is a formal refinement of ω_t that determines both an order in which transitions are applied, and an order in which occurrences are visited.

A central concept in this refined semantics is the *active constraint*. Each constraint becomes active immediately after it is added to the constraint store, and again each time a built-in constraint is added that further constrains one of its arguments (as this could cause guards to succeed). Each time a constraint becomes activate, all its occurrences are tried in an order fixed by the semantics, where rules are always tried in a top-down textual order. When a rule fires, its body is processed from left to right. Every new CHR constraint that is processed, is activated immediately. Every new built-in constraint is solved for, and all affected CHR constraints are activated one by one before processing the next constraint in the body. While processing a query or a body, ω_r treats CHR constraints as procedure calls: each time a constraint becomes active it searches for matching rules in order, until all matching rules have been executed, or the constraint is removed. As with a procedure, when a matching rule fires, other CHR constraints might become active, and, only when their execution finishes, the exectution returns to finding rules for the previous active constraint.

Extending the refined semantics with aggregates is of course also possible. Due to space considerations, we only outline the basic idea here. Analogous to the definition of the refined semantics for CHR$^{\neg}$ of [18, 19], constraint removals should be placed on the execution stack (next to constraint activations), and a new applicability condition has to be used that takes aggregates into account. This applicability condition can readily be based on Definitions 3 and 4 of Section 3.3.

4 Implementation

We have realized a prototype implementation as a source-to-source preprocessor for the K.U.Leuven CHR system [14, 15] in SWI-Prolog [20].

The preprocessor transforms a CHR program with aggregates into an equivalent regular CHR program. The latter can be dealt with by our existing CHR compiler[2]. The transformation introduces auxiliary constraint symbols and rules, and adds new heads, guards and body goals to existing rules. The process is similar to the one current CHR programmers go through, as illustrated in Section 2.1 under the *Common Approach*. The main difference is that now the preprocessor takes care of introducing all the cross-cutting code behind the scenes, not unlike an aspect weaver in Aspect-Oriented Programming [11].

The concrete implementation is based on high-level meta rules, which are discussed briefly in Section 4.1. The remaining sections gradually introduce the transformations applied for both supported computation schemes, *on-demand aggregate computation* (Sections 4.2 and 4.3), and *incremental aggregates computation* (Sections 4.4 till 4.6). Finally, Section 4.7 provides an overview of some implementation details and complications we encountered.

The correctness of the transformation schemes relies heavily on the execution order determined by the refined operational semantics, introduced in Section 3.4.

4.1 Meta CHR Rules

The source-to-source preprocessor is implemented using *meta CHR rules*. Meta CHR rules are designed specifically for the high-level declaration of source-to-source transformations of CHR programs. Due to space considerations, we only discuss the basic ideas behind meta rules here. This suffices for a good understanding of the illustrative examples used in subsequent sections.

Meta CHR rules closely resemble ordinary CHR rules, both syntactically and semantically. Only, instead of rewriting constraint multisets, they rewrite CHR rules of some *object CHR program*. A clear advantage of meta rules over previously proposed source-to-source frameworks for CHR [6], is that their high-level syntax allows for more concise declarations of source-to-source transformations.

The head of a meta CHR rule can contain *meta occurrences*. Operationally, these occurrences look for matching occurrences in a object rule's head. A meta rule is applicable if, for all its meta occurrences, matching object occurrences are found in a single object rule's head. When a meta rule fires, the head conjuncts that matched its removed meta occurrences, are removed from the object rule. Next, the body of the meta CHR rule is executed. These bodies can add conjuncts to the object rule's head, extra conjuncts to the object rule's guard (using the prefix operator `?/1`), or new rules to the object program. The concrete high-level syntax used should be self-explanatory.

[2] Some modifications to the CHR compiler were nevertheless necessary. These will be described in Section 4.7

4.2 Basic Compilation Scheme: passive aggregates

The core of the transformation is depicted by the meta rule in Figure 2. For compactness, pseudo code is used in these illustrative examples. The actual meta rules are very close to this pseudo code though.

The compilation scheme introduced here implements the *on-demand computation scheme* sketched in Section 2.2, where aggregates are recomputed from scratch when needed. The version of this section only deals with *passive aggregate computations* (cf. Section 2.2). The next section extends it to also account for *active aggregate triggerings*. The sections thereafter present an alternative computation scheme where aggregate results are *maintained incrementally*.

Compiling a non-maintained, passive aggregate is still relatively easy. The aggregate result simply has to be recomputed each time a matching is found for the remainder of the object head. This is realized by replacing each passive aggregate occurrence with a guard that computes and checks the aggregate (line 4). The auxiliary CHR constraint used is called `check`$_i$`(HV,A)`. Its first argument, `HV`, is a list of all variables occurring in the remainder of the head. These can be necessary to perform the matching with `G` (line 6). The second argument returns the aggregate's result `A`.

The `check`$_i$`/2` operation is implemented by the rule added to the object program on line 5. First, the aggregate's result is initialized using `Init`, and stored in the form of a `result`$_i$`/1` constraint. Then, a `match`$_i$`/1` constraint is added, which causes the result to be updated for each matching goal `G` (line 6). The actual incrementing of the `result`$_i$`(R)` constraint is done after an extra indirection through the auxiliary constraint `update`$_i$`(X)` (line 7). After all matchings are performed, the call to `match`$_i$`/1` returns, and the computed `result`$_i$`(R)` is retrieved from the constraint store. Lastly, the finalizer predicate `Final` is called, and the answer `A` is returned.

If at some point `Init`, `Inc` or `Final` fails, the `check`$_i$ guard also fails and the rule will not fire.

```
1  aggregate(Init,Inc,Final,_,X,G,A) # passive <=>
2   new_unique_identifier(i),
3   remaining_head_variables(HV),
4   ?checki(HV,A),
5   ( checki(HV,A) <=> Init(I), resulti(I), matchi(HV), geti(R), Final(R,A)),
6   ( matchi(HV), G ==> updatei(X) ),
7   ( updatei(X), resulti(R1) <=> Inc(R1, X, R2), resulti(R2) )
8   ( resulti(R), matchi(_), geti(Q) <=> Q = R ).
```

Fig. 2. Meta-rule that implements the translation for passive, non-maintained, non-optimized aggregates.

4.3 Active Aggregates: `active` and `removal` heads

By default, the CHR runtime only checks whether a rule is applicable at two occasions: when a CHR constraint matching one of its heads is added, or, when a built-in constraint is added that could allow its guard to succeed. However, a CHR rule containing aggregates also has to be triggered when the outcome of one of its aggregates changes. To accomplish this, we introduced two special types of heads to the left-hand side of a CHR rule. Asides from their regular head, a CHR rule can now have heads annotated with `active` or `removal`. Both types of head indicate extra conditions under which a rule has to be retried.

Active heads An *`active` head* is only perceived when the active constraint (cf. Section 3.4) matches one of its occurrences. In that case, matching partner constraints are searched for both the other occurrences of that active head, and the occurrences of the regular head of the rule. If these are found, the rule is fired (if, of course, the rule has not yet fired with the combination of constraints found). When the active constraint matches an occurrence of the same rule, but outside the active head, the active head behaves as if it was not specified.

In a sense, `active` occurrences are the complement of occurrences annotated with pragma `passive`: Passive occurrences never becomes active, they are only used to match with partner constraints when one of the regular occurrences is active. An active occurrence on the other hand is not used in the matching when a regular occurrence is active, its single purpose is to become active (and possibly to match with partner constraints if another active occurrence of the same active head is activated).

Removal heads While normal and `active` occurrences are activated when constraints matching it are *added*, occurrences of a *`removal` head* become active when CHR constraints matching it are *removed*. Upon activation, i.e. when a matching constraint is removed, matching partner constraints are searched for both the other occurrences of their removal head, and all regular occurrences.

Active aggregates Using these heads, supporting non-passive aggregates becomes easy. It suffices to add the following meta rule *after* the one of Figure 2:

```
aggregate(_,_,_,_,_,G,_)#Id ==> G#active, G#removal, pragma(passive(Id)).
```

The rule depicted in Figure 2 already removes all `passive` aggregates, *before* the above rule is tried. Therefore, the above rule fires only for non-passive aggregates. After adding the necessary heads to deal with the active aspects of the aggregate, this rule attaches a pragma `passive` to the aggregate, which causes the rule of Figure 2 to be applicable. Hence, active occurrences are passive occurrences, with some extra provisions to ensure correct triggering.

4.4 Incremental Aggregate Maintenance

The default strategy for computing aggregate values, explained in the previous two sections, is *on-demand*, i.e. *lazy*. Computing an aggregate that way has time complexity $\mathcal{O}(I + iG + M + F)$, where I, i, and F are the time complexities of the `Init`, `Inc`, and `Final` predicates respectively, G is the number of matches of `Goal`, and M is the time needed to find these matches. Usually, I, i, and F are $\mathcal{O}(1)$ and M is $\mathcal{O}(G)$, so the total complexity is $\mathcal{O}(G)$.

If G is large, and the aggregate has to be computed many times, then it is more efficient to maintain the aggregate value incrementally, rather than to recompute it from scratch every time it is needed. For this purpose, we have added the `#maintain` pragma (`#m` for short). When a rule containing an aggregate with pragma `#m` is first applied, the value of this aggregate is stored, and from then on, it is maintained dynamically. Each time a new matching is found for the `Goal` of the maintained aggregate, its stored value is updated accordingly by calling `Inc`. Conversely, if one of the constraints from such a matching is removed, the decrement predicate `Dec` is applied. Intuitively, `Dec` undoes an increment: i.e., if `Inc(A,X,B)`, then `Dec(B,X,A)`. However, `Dec` is allowed to fail, in which case we say the maintained value is invalidated. An invalidated value is not maintained incrementally any further. Instead, the next time the aggregate's value is needed, it will again be recomputed from scratch (and maintained again from then on).

Maintaining aggregate values reduces aggregate computations to cheap constant time lookups. Constraint insertions and removals however become slightly more expensive. Since sometimes the maintenance cost outweighs any performance gains, we left it to the programmer to declare when to maintain values.

In database literature, aggregate functions are classified as either *distributive*, *algebraic*, or *holistic* with respect to insertion or removal [12]. When an element is inserted or removed, *distributive aggregates* can be computed from the old aggregate value and the element. For example, `sum` and `count` are distributive for both insertion and removal, while `min` and `max` are distributive for insertion but not for removal. For distributive aggregates, `Final` is typically simply the identity function. *Algebraic aggregates* can be computed using a constant size working value. For example, `avg` and `stddev` are algebraic for both insertion and removal by maintaining the count and the sum (and the sum of squares). We exploit the algebraic property of `avg` by implementing it as follows:

```
:- chr_expansion avg(E, G, R)
     ---> aggregate(a_init, a_inc, a_dec, a_final, E, G, R).
a_init(t(0,0)).                    a_final(t(C,S), S/C) :- C>0.
a_inc(t(C,S), X, t(C+1,S+X)).      a_dec(t(C,S), X, t(C-1,S-X)).
```

Finally, holistic aggregates like `min` and `max` need, in general, a larger than constant-size working value for incremental computation. Our approach supports all three classes of aggregates because the working value can be arbitrarily large.

By allowing the decrement predicate `Dec` to fail, we can implement aggregates that are *sometimes* distributive for removal. Consider the following simple but often effective implementation of `min`/3:

```
:- chr_expansion min(E, G, R)
     ---> aggregate(m_init, m_inc, m_dec, m_final, E, G, R).
m_init(undef).                m_final(CM,CM) :- CM \= undef.
m_inc(CM,X,NM) :- (CM = undef ; X < CM) -> NM = X ; NM = CM.
m_dec(CM,X,CM) :- X \= CM.
```

When a new element arrives, the maintained minimum value is modified if the new element is smaller than the current minimum value. When an element is removed which is not the current minimum, the minimum is not affected. When the minimal element is removed, the decrement predicate fails and the minimum will be recomputed from scratch the next time it is needed. This way, `min` uses constant space and *sometimes* avoids recomputation even though it is holistic.

As an alternative implementation for `min`, the minimum can be maintained using a priority queue data structure like a Fibonacci heap. This results in a space overhead linear in the number of matches with the `Goal` and a logarithmic time overhead for removal. However, the minimum never needs to be recomputed, which may result in an overall time complexity improvement (cf. Section 5.2).

4.5 Incremental Aggregate Maintenance: Implementation

The basic compilation scheme for a maintained aggregate is given in Figure 3. The most important difference with the scheme depicted in Figure 2 is that when the result is retrieved from the constraint store, it is not removed (line 15). Instead the $\texttt{result}_i$ and $\texttt{match}_i$ constraints remain in the store[3]. As a result, all subsequent calls to $\texttt{check}_i$ with the matching head variables `HV` immediately return the precomputed result (line 8).

We now ensure the maintained result remains correct. Firstly, if a goal `G` that contributed to an aggregate's result is *removed*, the result has to be decremented accordingly. This can easily be accomplished using a `removal` head (lines 13 and 14), introduced by Section 4.3. If the `Dec` operation fails, the maintained result is invalidated, i.e., it is removed from the constraint store, such that the result will again be recomputed from scratch at next call to $\texttt{check}_i$ (lines 9–10).

Secondly, to keep an aggregate result consistent when *new matchings* for the aggregate goal `G` are found, the same rule can be used as before in Figure 2 (line 11). The propagation history of the rule ensures that each matching goal `G` only contributes to an aggregate's result once. This time, not only the $\texttt{match}_i/2$ occurrence can be active, but also the matched goal `G`. Therefore, it is important to add the rule *before* all other rules, to ensure a result is always updated before

[3] Note that because multiple $\texttt{result}_i$ constraints can now be in the store simultaneously, an extra identifier `Id` argument had to introduced to uniquely link the $\texttt{result}_i$ and $\texttt{match}_i$ couples.

```
1  aggregate(Init,Inc,Final,Dec,X,G,A)
2     # ( passive, maintain,
3         early_fail_when(Fail), early_succ_when(Succ) )
4   <=>
5     new_unique_identifier(i),
6     remaining_head_variables(HV),
7     ?check_i(A,HV),
8     ( result_i(Id,R), match_i(Id,HV) \ check_i(A,HV) <=> Final(R,A) ),
9     ( check_i(A,HV) <=> Init(I), result_i(Id,I), match_i(Id,HV),
10                       ( get_i(Id,R) -> Final(R,A) ; true ) ),
11    ( match_i(Id,HV), G ==> update_i(Id,X) ) - first,
12    ( update_i(Id,X), result_i(Id,A1) <=> Inc(A1, X, A2), result_i(Id,A2) ),
13    ( G#removal, match_i(Id,HV)#passive \ result_i(Id,A1)#passive
14         <=> (Dec(A1,X,A2) -> result_i(Id,A2) ; true) ) - first,
15    ( result_i(Id,A) \ get_i(Id,R) <=> R = A ),
16    ( get(_,_) <=> fail ),
17    ( Succ \ result_i(Id,A), match_i(Id,R) <=> true ),
18    ( Fail, result_i(_,A) ==> fail ).
19
20  aggregate(_,_,_,_,_,G,_)#Id ==> G#active, G#removal, pragma passive(Id).
```

Fig. 3. Meta-rules that implement the translation for maintained aggregates with the early failure and success optimizations.

it is checked (cf. Section 3.4). Hence the "`- first`" annotation. Also, if the rule was not inserted at the top of the object program, it would be possible that constraints of the newly matched goal are removed, before the corresponding increment was made. This would cause the result to be decremented for a goal that did not contribute to the result[4].

4.6 Optimizations

Figure 3 contains two simple optimizations, *early failure* and *early success*, that in certain cases avoid much unnecessary computation.

Early failure. Sometimes knowing the exact result is not necessary. The '`\+`' aggregate for instance is implemented as a `count` of the matching goals, followed by a guard which checks whether the result is zero. The exact number of matching

[4] Similar scenarios are still possible when constraints are triggered after a unification, because we have insufficient control over the order in which constraints are woken. This is a known problem, and a general solution exists. This solution is described in [18]. However, because this solution is overly verbose and very inefficient, and since it is only necessary for very rare scenarios, we choose not to include it in the current prototype transformation scheme. More effective solutions based on priorities are currently being developed [3].

goals is not important if it is not zero: it suffices to fail as soon as the first matching goal is encountered. More generally, if an aggregate's result grows monotonically in some direction, and there is a guard constraining the result in the opposite direction, the aggregate computation can safely fail as soon as this guard fails. For this purpose, an aggregate can be annotated with an *early failure condition* using the `early_fail` pragma. The implementation is straightforward: the rule added on line 18 states that we have to fail as soon as the early fail condition is met. Consider as a second example a (simplified) representation of a company and its employees as the constraints `company`/1 and `employee`/2. Let a small company be defined as a company with less than ten employees:

```
company(C), count(employee(_,C),Nb) ==> Nb < 10 | small(C).
```

Since large companies can have far more employees than ten, it is more efficient to write this as follows:

```
company(C), count(employee(_,C),Nb) # early_fail(?(Nb >= 10))
         ==> small(C).
```

The preprocessor already performs this kind of optimizations automatically, unburdening the programmer from doing this manually.

Early success. The converse can also be interesting: there are cases where one can safely *succeed* without computing the exact result. This can be indicated by pragma `early_succ`. When the *early success condition* is met (line 17), we stop incrementing the result. This causes the retrieval of the result to fail (line 16), which in turn causes the aggregate to succeed without finalizing and binding the answer (line 10). This makes perfect sense, as succeeding early implies that the result is not completely computed.

There are however some caveats when using this pragma. Consider for example the following rule:

```
company(C), count(employee(_,C),Nb) ==> Nb >= 100 | large(C).
```

This is transformed by the preprocessor to:

```
company(C), count(employee(_,C),Nb) # early_succ(?(Nb >= 100))
          ==> var(Nb) | large(C).
```

The `var(Nb)` guard is necessary to test whether the early success optimization was applied. If this guard were omitted, the rule would also fire if the count did succeed, i.e., with a result less then 100. When using the `early_succ` pragma, it is also important that the fully computed result is not needed somewhere further in the guard or in the body, as it will never be fully computed. Luckily though, the preprocessor is capable of detecting the most common cases automatically, so any explicit use of the pragma can be left to expert users.

We have also implemented two other simple refinements to reduce the number of maintained results and improve the number of cache hits. For simplicity, these are not included in the rules of Figure 3.

Selective variable passing. The compilation scheme of Figure 3 still uses a list `HV` of all variables of the remaining heads (line 7). This list probably includes variables that are not needed for performing the matching with `G` (line 11). In Figure 2, where the list was only used for this matching, these superfluous variables were not yet a problem. Here however, it also determines cache hits (line 8). Hence superfluous variables now lead to unnecessary cache misses, and consequently to more recomputations and larger caches. The actual generated code avoids this problem by only passing those variables that are really necessary to $\texttt{check}_i$. This way, both time and space complexity can be improved considerably.

Furthermore, on line 9, the preprocessor ensures that $\texttt{match}_i$ is only passed those variables necessary for matching (line 11), while $\texttt{result}_i$ receives those necessary for updating the result (line 12). This way, less variables have to passed to $\texttt{update}_i$, resulting in some extra time performance improvements.

Sharing aggregate results. Often the same aggregate occurs more than once in the same program. If a result has to be maintained for multiple compatible aggregate occurrences, it is cheaper to maintain only a single instance of this $\texttt{result}_i$. Whether this is possible depends on the element and the goal of the different occurrences, and also on the variables shared with the remainder of the head. This is a second reason care must be taken only to pass those variables that are really necessary (cf. supra).

4.7 Details and Complications

Fire-once versus fire-many semantics. CHR with aggregates is basically a generalization of CHR with negation as absence, mostly referred to as CHR$^{\neg}$. In [18, 19], a distinction is made between the fire-once, and the fire-many semantics for CHR$^{\neg}$. The *fire-many* semantics for CHR$^{\neg}$ essentially states that, if the negated head is satisfied more than once over time, a rule can also fire more than once for the same combination of constraints matching the non-aggregate occurrences. This can occur for instance when, after firing the rule, constraints matching the negated head are added and removed again. Whilst this made sense for CHR$^{\neg}$, we believe this is no longer the case in the more general aggregate setting. Hence, while [18, 19] opted for a fire-many semantics, in this work we opted for a *fire-once* semantics. In this semantics, the propagation history of the regular heads always prevents a rule to fire more than once with the same combination of constraints, as is the case in regular, aggregateless CHR.

This of course implies that the negation-as-absence aggregate, `\+`, has fire-once semantics, making it slightly different then the negated heads of CHR$^{\neg}$. However, we are confident this will hardly ever pose a problem in practice[5].

[5] Using pragma `no_history` it is sometimes possible to realize a fire-many semantics for aggregates, e.g., if all constraint arguments are ground. In the general case however, care must be taken as that this could allow the rule to fire too often.

Matching semantics. So far we have silently assumed that rules with aggregates are in some normal form, where the answer argument of an aggregate — i.e., the last argument of `aggregate/7` — is a variable not occurring elsewhere in the head, and possibly further constrained only by the guard of the rule. The semantics of using a non-variable as an answer argument, or a variable occurring elsewhere in the head, in fact involves transforming these cases into such a normal form. In other words, these cases correspond to implicit guards, made explicit by the preprocessor. One possibility would be to simply unify the actual result with the formal result argument. However, performing unifications in rule heads and guards is best avoided. Therefore, we opted for *one-way unification* or *matching*. A rule is thus applicable if the aggregates result *matches* the provided formal answer argument. This way, aggregates become analogous to regular occurrences, where matching semantics is also used.

Order of multiple aggregates. If more than one aggregate occurs in the same rule, the order in which the different aggregates are computed currently is left-to-right. Not only does it give some control on the order in which aggregates are evaluated, it also disambiguates the meaning of expressions like "`min(X, c(X), Y), max(Y, c(Y), X)`" in the head of a rule: to match the goal of an aggregate, variables that occur in the answer of another aggregate are also used, but only those of aggregates to the left of the aggregate. Note that variables occurring in the element or the goal part are local to the aggregate, and are thus always ignored by other aggregates. The above (nonsensical) expression is therefore equivalent to "`min(X, c(X), Y), max(A, (c(A), ?(A == Y)), X)`", and not for instance "`max(Y, c(Y), X), min(A, (c(A), ?(A == X)), Y)`".

Compiler changes required to deal with CHR constraints in guards. Aggregates are transformed into calls to CHR constraints in guards. Though not uncommon, this is actually not allowed in CHR. As seen in Section 3.1, guards are only really meant to contain built-in constraints. Both the compilation schema of the K.U.Leuven CHR compiler, and several of its optimizations and analyses are based on this restriction. The CHR constraints, added to the guard by our preprocessor, inspect the constraint store for the presence of constraints matching the aggregate's goal. For efficiency reasons though, the K.U.Leuven CHR compiler defers the allocation and storage of constraints as long as possible. The analyses responsible to determine this, however, did not take such constraint-store-observing guards into account. Therefore it was possible that constraints that were supposed to be used in matchings for the aggregate's goal, were not yet added to the constraint store, or even not yet allocated. Consequently, not always were all required matchings for the aggregate's goal considered.

Hence, the compiler had to be modified to better deal with CHR constraints in the guard[6]. We rewrote the *observation analysis* such it takes into account

[6] For backwards compatibility reasons, all changes to the compiler described in this paragraph only apply when the `store_in_guards` option is switched on. Otherwise, all rewritten analyses behave completely the same as before.

constraints observed by guards. The *late allocation analysis* had to be rewritten, as guards can now necessitate constraint allocations as well. An extra complication encountered here was that the only safe allocation points in the current compilation scheme are at kept occurrences, even though now it could be necessary to allocate at removed occurrences also (as the constraints have to be stored shortly for observation by the guard; immediately after the guard they will be removed again). Finally, *code generation* was adjusted to allow constraint storage prior to guards, rather then at the beginning of bodies. Unlike before, constraint storage is now also possible for removed occurrences.

Note that these changes are not only useful in the context of aggregates, they can also improve the behavior of other programs that use CHR constraints in guards.

Failing guards lead to failing aggregate maintenance. Transforming aggregates to guards has its downside. If some later guard conjunct fails, the runtime backtracks over the aggregate's computation, and, more importantly, over storing the result if the aggregate had to be maintained. Consider for example the following rule:

```
company(C), nb(employee(_,C),Nb)#maintain ==> Nb > 15000 | large(C).
```

For the first 15.001 employees hired, the number of employees is always recounted from scratch, because each time the guard failed, causing the maintained result to disappear due to backtracking. So for aggregates that have to be maintained, but whose result is further constrained by a later guard, or if there are other guards that can subsequently fail, the current compilation scheme is not ideal. We currently provide a rather ad-hoc solution: it is possible to declare that an aggregate has to be maintained in the body of a rule as well. Using this possibility, the above rule can be rewritten to:

```
company(C) ==> maintain( nb(employee(_,C), _) ).
company(C), nb(employee(_,C), Nb) ==> Nb > 15000 | large(C).
```

Note in passing that this feature also allows for a more fine-grained control of which aggregates have to be maintained.

The problem sketched above can always be solved by adding a propagation rule that explicitly states, in its body, that the aggregate's maintenance has to be started. As a point for future work, a compilation scheme could be investigated that performs similar changes automatically. Another alternative is not to transform an aggregate to a guard, but to keep aggregates in the rule's head. The latter compilation scheme would in fact be closer to the one outlined in Section 2.1 under *Common approach*.

Implementation of `active` and `removal` heads using pragma `history`. The `active` and `removal` heads introduced in Section 4.3, are implemented as a second source-to-source transformation. The rules containing these special

heads are duplicated several times, and pragma `passive` is used extensively to get the desired semantics. Triggering `removal` occurrences on constraint removal is realized by adding extra auxiliary constraints in the beginning of the bodies of the rules that remove constraints occurring in `removal` heads.

This approach was chosen over a direct implementation of the new heads in the compiler, because this way all existing analyses could remain unadjusted. Only one extra change was required to the compiler, namely, a new pragma had to be added to enable sharing of propagation histories: if the source-to-source transformation duplicates a propagation rule, all duplicates have to use the same history. The pragma is called `history`/2. The first argument is a ground *history identifier*, uniquely identifying the shared history. The second argument is a list of *occurrence identifiers*. The occurrences these identifiers designate in the different instances of the same shared history should be occurrences of corresponding constraints. The following example will clarify:

```
a#A, b#B ==> d pragma history(my_history, [A,B]).
a#A, b#B, c ==> e pragma history(my_history, [A,B]).
```

Adding this pragma to the compiler was relatively easy. It certainly required less work than adding `active` and `removal` heads directly. Also, pragma `history`/2 has already proven its usefulness outside the context of this work.

Nested aggregates. The implementation of non-passive aggregates that contain nested aggregates in their goal required extra work, as the result of the outer aggregate can change if the result of a nested aggregate changes. For now, suppose the aggregates, both the outer and the inner, do not have to be maintained incrementally. If `G` contains nested aggregates, the lazy compilation scheme of Section 4.3 generates aggregates part of `active` and `removal` heads. As aggregates are not really removed, aggregates part of a `removal` head can be discarded. However, aggregates do get active, namely when matchings arrive or disappear for their goal. Therefore, for aggregates in `active` heads, the necessary `active` and `removal` heads, based on their goals, have to be generated.

Nested aggregates can be maintained. The prototype implementation however cannot yet maintain an outer aggregate, i.e. an aggregate that aggregates over one or more nested aggregates. The current transformation scheme can only ensure that increments are made when a nested aggregate's value changes, but not that first a decrement is made with the old aggregate value. This minor limitation of the current prototype implementation will be addressed in future work.

5 Evaluation

In this section we evaluate our implementation of aggregates in two ways. First we evaluate the additional expressive power by comparing programs written with and without aggregates. Then we measure and compare running times.

```
:- chr_constraint  state(+), delta(+,+,+),
   input(+), final(+), start, loop, b(+,+),
   a(+,+,+), l(+,+), k(+), p(+,+), mov(+,+),
   c(+,+), new_l(+,+,+), inv(+,+), fix(+,+),
   add_a(+,+,+), nb_a(+,+,+).
start, final(S) \ state(S) <=> b(1,S).
start           \ state(S) <=> b(2,S).
b(I,S), input(A) ==> nb_a(A,I,0),add_a(A,I,S).
delta(_,A,S) \ add_a(A,I,S) <=> a(A,I,S).
add_a(_,_,_) <=> true.
a(A,I,S) ==> nb_a(A,I,1).
nb_a(A,I,X), nb_a(A,I,Y) <=> nb_a(A,I,X+Y).
start ==> k(3).
start, input(A) ==> new_l(A,1,2).
nb_a(A,J,X), nb_a(A,K,Y)
 \ new_l(A,J,K) <=> X =< Y -> l(A,J) ; l(A,K).
start <=> loop.
loop, l(A,I) <=> p(A,I).
loop         <=> true.
p(A,I), a(A,I,X) ==> inv(A,X).
inv(A,X), delta(T,A,X) \ b(J,T) <=> c(J,T).
inv(_A,_X) <=> true.
p(_,_), c(J,_)#p \ k(K) <=> mov(J,K).
mov(J,K) \ c(J,T)#p <=> b(K,T), fix(J,T).
fix(J,T) \ a(A,J,T) <=> nb_a(A,J,-1).
fix(_,_) <=> true.
mov(J,K), input(A) ==> new_l(A,K,J).
mov(_,K) <=> k(K+1).
p(_,_) <=> loop.
```

```
:- chr_constraint  state(+), delta(+,+,+),
   input(+), final(+), start, loop, b(+,+),
   a(+,+,+), l(+,+), k(+), p(+,+), mov(+,+),
   c(+,+), new_l(+,+,+), inv(+,+), fix(+,+).

start \ state(S), nb(final(S),B) <=> b(2-B,S).

b(I,S), input(A), exists(delta(_,A,S)) ==> a(A,I,S).




start ==> k(3).
start, input(A) ==> new_l(A,1,2).
nb(a(A,J,_),X)#m, nb(a(A,K,_),Y)#m
 \ new_l(A,J,K) <=> X =< Y -> l(A,J) ; l(A,K).
start <=> loop.
loop, l(A,I) <=> p(A,I).
loop         <=> true.
p(A,I), a(A,I,X) ==> inv(A,X).
inv(A,X), delta(T,A,X) \ b(J,T) <=> c(J,T).
inv(_A,_X) <=> true.
p(_,_), c(J,_)#p \ k(K) <=> mov(J,K).
mov(J,K) \ c(J,T)#p <=> b(K,T), fix(J,T).
fix(J,T) \ a(A,J,T) <=> true.
fix(_,_) <=> true.
mov(J,K), input(A) ==> new_l(A,K,J).
mov(_,K) <=> k(K+1).
p(_,_) <=> loop.
```

Fig. 4. The programs HOPCROFT-orig and HOPCROFT-agg: implementations of Hopcroft's algorithm for minimizing states in a finite automaton [8].

5.1 Expressivity Case Studies

Figures 4, 5, 6, and 7 contain different versions of, respectively, the HOPCROFT, DIJKSTRA, EULER, and SUDOKU programs. On the left are versions without aggregates, on the right are versions which use, respectively, the aggregates `nb` and `exists`, `argmin` and `no`, `forall` and `nb`, and `takemin` and `nb`. We have also included figures for the BANKING example of Section 2.1. In all cases, the program that uses aggregates is more concise. We can quantify the gained conciseness in terms of the number of constraints, rules, and bytes in the program. Table 2 lists these numbers. In some cases, aggregates halve the program size w.r.t. to any of these metrics.

5.2 Evaluation

We expect programs that use aggregates to be slower than the original programs: our implementation has to deal with all possible use patterns of aggregate expressions, while the original programs are manually specialized. However, our implementation of aggregates would not be practical if programs that use aggregates are slower by more than a (reasonably small) constant factor. Figure 8 shows a plot of benchmark results for different versions of the DIJKSTRA, EULER, and HOPCROFT programs. For SUDOKU we did not find a scalable benchmark, but preliminary results are similar to the other results. The n^2 and $n \log n$ lines

```
:- chr_constraint edge(+,+,+), dijkstra(+),
        d(+,+), scan(+), l(+,+), try(+,+).
dijkstra(A) <=> l(A,0), scan(A).

scan(A) \ l(A,D) <=> d(A,D).
scan(A), d(A,D), edge(A,B,W) ==> try(B,D+W).
d(B,_) \ try(B,L) <=> true.
l(B,X) \ try(B,L) <=> L >= X | true.
l(B,X) , try(B,L) <=> l(B,L), decr_key(B,L).
         try(B,L) <=> l(B,L), insert(B,L).
scan(A) <=> extract_min(B,_) | scan(B).
scan(_) <=> true.

% + some implementation of a priority queue
% with the operations insert(+,+),
% decr_key(+,+), and extract_min(-,-).
```

```
:- chr_constraint edge(+,+,+), dijkstra(+),
        d(+,+), scan(+), l(+,+).
dijkstra(A) <=> l(A,0), scan(A).
l(A,X) \ l(A,Y) <=> X =< Y | true.
scan(A) \ l(A,D) <=> d(A,D).
scan(A), d(A,D), edge(A,B,W), no(d(B,_))
        ==> l(B,D+W).


scan(A), argmin(L,l(B,L))#m#p <=> scan(B).
scan(_) <=> true.
```

Fig. 5. The programs DIJKSTRA-orig and DIJKSTRA-agg: implementations of Dijkstra's single-source shortest path algorithm.

```
:- chr_constraint node(+), edge(+,+), euler,
          test(+), degree(+,+), get_d(+,?).
euler, node(N) ==> test(N).
euler <=> true.
test(N), edge(N,_) ==> degree(in, 1).
test(N), edge(_,N) ==> degree(out,1).
test(N) <=> get_d(in,X), get_d(out,X).
degree(X,Y), degree(X,Z) <=> degree(X,Y+Z).
get_d(X,Q), degree(X,Y) <=> Q = Y.
get_d(X,Q) <=> Q = 0.
```

```
:- chr_constraint node(+), edge(+,+), euler.

euler, forall(node(N),
         (
           nb(edge(N,_),X),
           nb(edge(_,N),X)
         )
       )#p <=> true.
euler <=> fail.
```

Fig. 6. The programs EULER-orig and EULER-agg: is a connected digraph Eulerian?

```
:- chr_constraint solve, v(+,+), c(+,+),
                  nb_c(+,+), solve(+).
solve <=> solve(1).
solve(N), c(P,V), nb_c(P,N)
  <=> (v(P,V) ; N>1, nb_c(P,N-1)), solve(1).
solve(N) <=> N<9 | solve(N+1).
solve(_) <=> true.
c(P,_) ==> nb_c(P,1).
nb_c(P,X), nb_c(P,Y) <=> nb_c(P,X+Y).
v(P,_) \ nb_c(P,_) <=> true.
v(P,_) \ c(P,_) <=> true.
v(P,V) \ c(Q,V), nb_c(Q,N)
    <=> row_col_box(P,Q) | N>1, nb_c(Q,N-1).
```

```
:- chr_constraint solve, v(+,+), c(+,+).


solve, takemin(N,(c(P,V),nb(c(P,_),N)))
    <=> (v(P,V) ; N>1), solve.

solve <=> true.



v(P,_) \ c(P,_) <=> true.
v(P,V) \ c(Q,V), nb(c(Q,_),N)
    <=> row_col_box(P,Q) | N>1.
```

Fig. 7. The programs SUDOKU-orig and SUDOKU-agg: solvers for *Sudoku* puzzles.

	original			aggregates			% gained		
Program	C	R	B	C	R	B	C	R	B
BANKING (Sec.2.1)	6	4	493	5	3	300	17%	25%	39%
HOPCROFT (Fig.4)	18	23	928	16	18	753	11%	22%	19%
DIJKSTRA (Fig.5)	6	9	386	5	6	281	17%	33%	27%
EULER (Fig.6)	6	8	338	3	2	131	50%	75%	61%
SUDOKU (Fig.7)	5	9	385	3	4	207	40%	56%	46%

Table 2. Expressivity gained by using aggregates, in terms of the number of constraints (C), the number of rules (R), and the number of bytes (B).

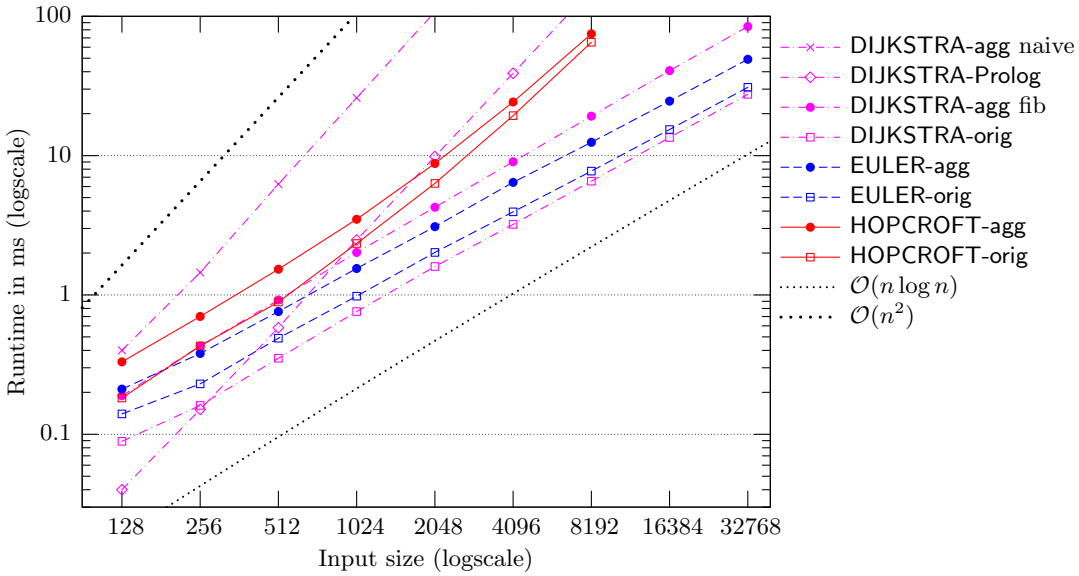

Fig. 8. Plot of the execution times for different versions of the the DIJKSTRA, EULER, and HOPCROFT programs.

are shown for ease of reference. Firstly, note that for the DIJKSTRA-agg program, the naive implementation of `argmin` results in an $\mathcal{O}(n^2)$ time complexity (just like Dijkstra-Prolog[7]) while the Fibonacci heap implementation results in the optimal $\mathcal{O}(n \log n)$ time complexity. All CHR programs except the naive version of DIJKSTRA-agg have the correct time complexity. The (non-naive) DIJKSTRA-agg program is about three times as slow as the manually specialized DIJKSTRA-orig program. Part of this constant overhead is caused by the implementation of Fibonacci heaps: the implementation of `argmin` potentially needs multiple heaps, while the version used in DIJKSTRA-orig is specialized to just one heap. In addition, DIJKSTRA-orig uses an update operation (`decr_key`) where our implementation of aggregates uses only delete and insert operations. For the EULER and HOPCROFT programs, the version without aggregates is only about 1.5 times faster than the versions with aggregates.

6 Related Work

Constructs related to aggregates are found in many languages. In this section we briefly discuss some of them.

SQL. For SQL [2], the standard query language for databases, which unlike CHR [17] is not Turing-complete, aggregates are very important because they

[7] a Prolog implementation of Dijkstra's shortest path algorithm by Roman Barták. Available at *http://ktiml.mff.cuni.cz/~bartak/prolog/graphs.html*.

add computational power to the language. The original SQL standard only supports five aggregate functions: `min`, `max`, `count`, `sum`, and `avg`. However, practice showed that users often require to aggregate data in many other ways. To meet this need, all major database systems added numerous other built-in aggregate functions, some of which were standardized in later revisions of the SQL standard[8]. More recently, many database systems also include the possibility to extend the database query language with user-defined aggregates.

Production Rule Systems. The first generation of production rule (PR) systems offered only limited support for aggregates: most systems had negation as absence (`\+`), some also allowed `exists` and `forall`. Upcoming versions of several major PR systems will introduce a general `accumulate` construct, similar to our `aggregate`/7. However, to the best of our knowledge, no PR system provides syntactic shorthands for commonly used aggregates (cf. Table 1). Also, as far as we know, no PR system offers incremental aggregate maintenance.

CHR$^\neg$. The `\+` aggregate is similar to negation as absence in CHR$^\neg$ [19]: there are some minor differences, one of which is explained in Section 4.7, but CHR with aggregates can be considered as a generalization of CHR$^\neg$.

Logic Programming. Semantics of aggregates have been widely studied in the context of logic programming [10, 13]. The best-known practical implementation of aggregates are the *all solutions* predicates `findall/3`, `bagof/3` and `setof/3` of ISO-Prolog [1]. The latter two are nondeterministic, and hence correspond to aggregate *relations*, rather than the usual deterministic aggregate *functions*. Other aggregates can be implemented in terms of these all solutions predicates.

Functional Programming. Aggregates are a special case of *catamorphisms* or *folds* (a.k.a. reduce) from category theory, which are widely applied in functional programming. While aggregates usually consider implicit and unstructured collections of data (sets and multi-sets), catamorphisms deal with explicit and tree-shaped algebraic data structures. Many laws for fold have been established using various equational reasoning techniques, e.g. for the parallel (or incremental) computation of folds [9].

Object-Oriented Programming. Object-oriented (and other) imperative languages often deal with aggregates in a low-level way: by explicit iteration over collection objects. The *iterator* design pattern [7] captures this concept. However, more and more, the need for a higher-level syntax becomes apparent. OOP language designers increasingly turn towards declarative languages for a solution. For example the C# extension LinQ[9] offers a SQL-like syntax for querying data structures. However, the underlying implementation consists of various higher-order functions, e.g. the generic `Aggregate` function is really a fold.

[8] e.g. `every` and `any/some` in SQL-99 and several statistical aggregates in SQL-2003
[9] See `http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx`.

7 Conclusion and Future Work

In this paper we have proposed aggregates as a new language feature for CHR. We have argued that it considerably increases the expressiveness of CHR and reduces cross-cutting code, as illustrated in a number of case studies. Our proposal concerns a general aggregates infrastructure that not only allows for a rich set of predefined aggregates, but also caters for user-defined application-specific aggregates. We have provided a first implementation as a source-to-source transformation. Benchmarks indicate that the desired complexity is attainable.

In future work, various ways can be investigated to improve the efficiency of our aggregates implementation. In particular, both specializations on the source level, as well as dedicated support in the CHR compiler can be considered. Incremental maintenance of aggregates can e.g. be embedded directly in the constraint store insertion and removal operations. It may also be interesting to investigate alternative source-to-source transformation schemes.

With the arrival of this new high-level language feature, CHR-programmers are faced with the challenge of updating their existing code to the new quality standard. For this purpose, semi-automatic refactoring support in the style of [16] can be developed.

A final topic for future work is the development of static (and perhaps also dynamic) analyses to automatically select the aggregate computation strategy: on-demand or incremental, or maybe even hybrid strategies.

Acknowledgments

Part of this work was done while Jon Sneyers was visiting the University of Melbourne and the NICTA Victoria Lab. We thank Peter J. Stuckey and Gregory J. Duck for the many discussions and valuable input in the initial stages of this research. We are grateful to Leslie De Koninck for his supporting work on the K.U.Leuven CHR compiler.

Jon Sneyers is funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen). Peter Van Weert is a Research Assistant of the fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen). Tom Schrijvers is a Post-Doctoral Researcher of the fund for Scientific Research - Flanders (Belgium).

References

1. ISO/IEC 13211:1995: Information technology – Programming languages – Prolog.
2. ISO/IEC 9075:2003: Information technology – Database languages – SQL.
3. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. CHR^{rp}: Constraint handling rules with rule priorties. Technical Report CW481, Department Computer Science, K.U.Leuven, Belgium, March 2007.

4. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In Bart Demoen and Vladimir Lifschitz, editors, *20th International Conference on Logic Programming (ICLP'04)*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104, Saint-Malo, France, September 2004. Springer Verlag.
5. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
6. Thom Frühwirth and Christian Holzbaur. Source-to-source transformation for a class of expressive rules. In *Joint Conference on Declarative Programming APPIA-GULP-PRODE 2003 (AGP 2003)*, Reggio Calabria, Italy, September 2003.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
8. John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. Technical Report STAN-CS-71-190, Stanford University, CA, USA, 1971.
9. Zhenjiang Hu and Masato Takeichi. A calculational framework for parallelization of sequential programs. In *International Symposium on Information Systems and Technologies for Network Society*, pages 102–109, Fukuoka, Japan, September 1997.
10. David B. Kemp and Peter J. Stuckey. Semantics of logic programs with aggregates. In *Intl. Symp. Logic Programming*, pages 387–404, San Diego, USA, 1991.
11. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Meda, Christina Lopes, Jean-Marc Loingtier, and John Irwing. Aspect oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*. LNCS 1241, 1997.
12. Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *28th International Conference on Very Large Data Bases*, Hong Kong, China, 2002.
13. Nicolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming*, 2007. To appear.
14. Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, June 2005.
15. Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In Thom Frühwirth and Marc Meister, editors, *Selected Contributions, 1st Workshop on Constraint Handling Rules*, Ulm, Germany, May 2004.
16. Alexander Serebrenik, Tom Schrijvers, and Bart Demoen. Improving Prolog programs: Refactoring for Prolog. *Theory and Practice of Logic Programming*, 2007. To appear.
17. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. In *2nd Workshop on Constraint Handling Rules*, pages 3–17, Sitges, Spain, October 2005.
18. Peter Van Weert, Jon Sneyers, Tom Schrijvers, and Bart Demoen. CHR$^{\neg}$: Constraint Handling Rules with Negation. Technical Report CW446, Dept. of Computer Science, K.U.Leuven, Belgium, May 2006.
19. Peter Van Weert, Jon Sneyers, Tom Schrijvers, and Bart Demoen. Extending CHR with negation as absence. In *3rd Workshop on Constraint Handling Rules*, pages 125–139, Venice, Italy, 2006.
20. Jan Wielemaker. An overview of the SWI-Prolog programming environment. In *13th Intl. Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, 2003. Home page at *http://www.swi-prolog.org*.