Aggregates for Constraint Handling Rules

Jon Sneyers*, Peter Van Weert**, and Tom Schrijvers***

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

Abstract. We extend the Constraint Handling Rules language with aggregates such as `sum`, `count`, `findall`, and `min`. The proposed extension features nested aggregate expressions over guarded conjunctions of constraints, a series of predefined aggregates, and application-tailored user-defined aggregates. We formally define the operational semantics of aggregates, and show how incremental aggregate computation facilitates efficient implementations. Case studies demonstrate that language support for aggregates significantly reduces program size, thus improving readability and maintainability considerably.

1 Introduction

Constraint Handling Rules (CHR) [6, 4] is a powerful, elegant committed-choice CLP language, based on multi-headed, guarded multiset rewrite rules. Originally designed for the implementation of constraint solvers, CHR has matured towards a general purpose language, used in a wide range of application domains, including natural language processing, multi-agent systems, and type system design.

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. As the head of a single CHR rule considers only a *bounded* number of constraints, any form of aggregation necessarily requires explicit encodings using auxiliary CHR constraints and rules. Such ad hoc approaches are repetitive, cumbersome and error-prone, and the resulting auxiliary constructs often cross-cut the entire program. In other words, all aforementioned advantages of declarative programming are handicapped severely.

With language support for aggregates, the programmer can express the aggregate logic concisely in a single self-contained rule, exhibiting all practical

* Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).
** Research Assistant of the Research Foundation – Flanders (FWO-Vlaanderen).
*** Post-Doctoral Researcher of the Research Foundation–Flanders (FWO-Vlaanderen).

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. However, information often has to be aggregated in application-specific ways. Our aggregate framework therefore caters for user-defined aggregates. Moreover, in our proposal, no restrictions are placed on aggregate expressions: aggregates may range over arbitrary guarded conjunctions of constraints; even nested aggregates are allowed.

The generic aggregate language construct is designed to allow efficient implementations using either an on-demand or an incremental computation strategy. Our initial prototype implementation shows the desired runtime complexity is attainable, with only a modest overhead compared to manually specialized code.

Overview. The rest of this paper is structured as follows. In Section 2 we motivate and introduce our new language feature. Next, Section 3 provides a formal definition of aggregates in CHR. Section 4 shows aggregates can be implemented efficiently using incremental computation, and briefly discusses our prototype implementation. In Section 5 we use a number of case studies to evaluate the performance of this prototype, as well as the expressiveness gained by the use of aggregates. Finally, Section 6 reviews related work and Section 7 concludes.

2 Aggregates

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 expressiveness, 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).
```

Clients however are allowed to have multiple accounts, so the bank later changes the business rule to: "*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 were limited to some fixed number n, all possible cases could be enumerated 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 replication of code exhibited by this naive approach is highly undesirable, as it is hard to read and maintain. This approach also scales badly performance-wise, as the number of combinations tried during matching increases exponentially. Last but not least, exhaustively enumerating all possible cases is impossible if n is unbounded.

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

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

Even though it concisely captures the logic of platinum in a single rule, and facilitates an unbounded number of accounts per client, this approach remains inadequate. The reason is that it necessitates the accumulated balance to be maintained *explicitly*. This inherently cross-cutting concern requires invasive modifications to all parts of the original code that alter the balance of an account. The following rules, spread throughout the entire program, have to be adjusted to update `accumulated_balance` accordingly (the underlined code is added):

```
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 modifications scattered throughout the entire program. Similar cumbersome auxiliary code has to be introduced for every aggregation, and, conversely, when adding new rules all existing aggregations must be taken into account. Clearly, this approach displays poor compliance with common software quality criteria: it is highly 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. Using an aggregate (in italics), the 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: it accumulates the sum of the balances `B` of all matching `account/4` constraints, and ensures that the rule fires as soon as this sum, `Sum`, reaches 25,000.

As a result, the program is more declarative, readable and maintainable. Relieved from the cumbersome and repetitive task of implementing aggregates, the programmer can focus exclusively on the application domain. So, productivity is improved as well.

Aggregate	*Meaning*	*Synonyms*
`nb(`G`,`C`)`	G matches C times	`count`
`collect(`X`,`G`,`L`)`	L is a list of X's for every match of G	`findall`
`exists(`G`)`	at least one matching of G exists	
`\+` G	negation as absence: no matching of G exists	`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. The template X is an arbitrary term. Instantiations of the template, based on the different matchings of the goal, are used to increment and decrement the aggregates value (cf. infra). This template is similar to the first argument of the well-known `findall`/3 ISO Prolog predicate [1]. For all arithmetic aggregates, X must evaluate to a ground arithmetic expression at runtime.

2.2 Syntax and Informal Semantics

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

Predefined aggregates. Table 1 lists the proposed predefined aggregates. Section 2.1 already showed an example of the `sum/3` aggregate; the other arithmetic aggregates are analogous. The following example shows the use of the `exists/1`. Several more examples of aggregates can be found in the case studies of Section 5.

Example 1. Consider the following rule:

```
client(C), account(_,C,savings,_) ==> ...
```

As clients can have multiple saving accounts, this rule may fire multiple times for the same client. If this is not the desired behavior, and the action in the body should not be performed more than once per client, then `exists/1` can be used:

```
client(C), exists(account(_,C,savings,_)) ==> ...
```

Complex aggregate goals and nested aggregates. So far we have only shown examples of aggregates over a simple goal, i.e., consisting of a single CHR constraint. More complex aggregate goals are also supported. 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 expressiveness 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)))`.

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

```
aggregate(Start, Inc, Dec, Final, Template, Goal, Result)
```

The function of the arguments is as follows:

`Start/1`	returns an initial working value;
`Inc/3`	takes a working value and an template instance and returns a new, incremented working value;
`Dec/3`	is the inverse of `Inc`, returning a decremented value;
`Final/2`	takes a working value and returns the result;
`Template`	is a template to describe an element for a given `Goal`;
`Goal`	is a conjunction of CHR constraints, guards, and nested aggregates;
`Result`	is the result of the aggregate.

The first four arguments refer to the host-language procedures or CHR constraints that compute the aggregate. This paper uses Prolog as a host-language, but our approach is applicable to any host-language. The last three arguments correspond to the arguments of the predefined arguments of Table 1. The semantics of the arguments is further explained below, and more formally in Section 3.

The general `aggregate/7` construct is expressive enough to formulate any aggregate function. In fact, all predefined aggregates are implemented through it. The next paragraph describes a first example. More detailed information on the different types of aggregates that can be expressed is provided in Section 4.1.

Informal semantics. Consider as a running example `sum(T,G,S)`. This aggregate is effectively implemented as `aggregate(=(0),plus,minus,=,T,G,S)`, where '`=(0)`' indicates unification with zero[1], and the Prolog predicates `plus/3` and `minus/3` compute the sum resp. the difference of their first two arguments. First, the working value V_0 of the aggregate is initialized by calling `Start`(V_0). In the case of `sum`, this results in an initial working value $V_0 = 0$. This value is then incremented once for each of the n matchings of `Goal`, by calling `Inc`(V_{i-1}, `Template`, V_i) for $1 \leq i \leq n$. For `sum`, the increment predicate `plus` adds the working value and the instantiated value of `Template`. Finally, V_n is finalized by calling `Final`(V_n, `Result`). Often, as with `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

[1] Extra arguments are appended to the provided predicates, so e.g. `Start`(V_0) with `Start` equal to '`=(0)`' becomes the unification `=(0,`V_0`)` (usually written infix: `0=`V_0).

`avg` are undefined when there are no elements. The decrement predicate `Dec` is discussed in Section 4.1, which describes an alternative computation strategy.

Operationally, a rule containing an aggregate is tried when one of the other head constraints is inserted or reactivated. 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, reactivated[2], or removed. This constitutes an *active* aggregate computation. In the banking example of Section 2.1, there is a passive aggregate computation when a new `client/3` constraint is added, and an active computation when an `account/4` constraint is added, reactivated or removed.

Syntactic shortcuts. Because the generic `aggregate/7` notation is clearly overly verbose, it is quite useful to use a macro facility for defining abbreviations. The CHR host language may already offer such a facility, e.g. term expansion in Prolog, or C's macro language. Nevertheless, we prefer to integrate some macro facility in the CHR language itself for portability reasons and possible scheduling conflicts with CHR compilation[3]. The syntax is as follows:

`:- chr_expansion` $atom_1$ `--->` $atom_2$`.`

This replaces any occurrence of $atom_1$ in the head of a rule with $atom_2$. 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 the naming of new aggregates and even the allocation of 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).
```

Such user-defined aggregate names further increase the readability and maintainability of a program.

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 [4, 6, 14].

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 *built-in constraints*. The host language considered in this paper is Prolog. Its data types are Prolog variables

[2] A constraint is *reactivated* if a built-in constraint is added that further constrains one of its arguments: cf. [4] for more information.

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

and terms, its built-in constraints unification, Prolog built-ins and other (user-defined) Prolog predicates.

CHR constraint symbols are predicate symbols, denoted by a functor/arity pair. *CHR constraints* are atoms constructed from these symbols and host language data types. A CHR program $\mathcal{P}$ is a set of CHR rules of the form:

$$name \ @ \ \ H_k \setminus H_r \iff G \mid B$$

where H_k and H_r are conjunctions of CHR constraints called the *heads* (*kept* and *removed* heads respectively), 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* | ".

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 " $\iff$ ". 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: $G \rightarrow (H_k \rightarrow B)$. The next section reviews CHR's operational semantics.

3.2 The Operational Semantics ω_t

Informally, the operational semantics of a CHR rule is as follows: if for each head a matching constraint is in the constraint store, and the guard is satisfied, then the constraints that matched the removed heads are deleted from the store and the body is executed. Formally, the execution of a CHR program follows the *theoretical* or *high-level* operational semantics [4, 6], commonly denoted as ω_t. 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 (Identified constraints). *To differentiate amongst otherwise identical copies of constraints, CHR constraints are assigned unique identifiers. An* identified *CHR constraint with* constraint identifier i *is denoted* $c\#i$*. We further 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.*

Definition 2 (Execution state). *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. The CHR constraint* store $\mathbb{S}$ *is a set of identified CHR constraints (while* $\mathbb{S}$ *is a set,* $chr(\mathbb{S})$ *is a multiset). The* built-in constraint store $\mathbb{B}$ *is the conjunction of all built-in constraints passed to the underlying solver. The* propagation history $\mathbb{T}$*, necessary to prevent trivial non-termination, is a set of tuples, each recording the name of a rule and a sequence of identities of the CHR constraints that fired that rule. Finally, the integer counter* n *represents the next unique constraint identifier.*

The semantics of the built-in constraints is determined by a constraint theory $\mathcal{D}_\mathcal{B}$. Let $vars(A)$ be the variables occurring freely in A, then $\bar{\exists}_A F$ denotes $\exists x_1, \ldots, \exists x_n F$, with $\{x_1, \ldots, x_n\} = vars(F) \backslash vars(A)$.

Definition 3 (Matching). *The set of* matching substitutions *is defined as:*

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

where H and S_h are conjunctions of CHR constraints, and G and $\mathbb{B}$ conjunctions of built-in constraints.

Definition 4 (Transition rules). *Given a CHR program $\mathcal{P}$, execution proceeds by exhaustively applying the following transition rules, starting from an* initial state *of the form $\langle \mathbb{G}, \emptyset, true, \emptyset\rangle_1$:*

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

3.3 Adding Aggregates to ω_t

We modify the definition of ω_t to deal with the `aggregate`/7-expressions introduced in Section 2.2 (recall this also indirectly covers all predefined aggregates). Because aggregates can be nested, we use two mutually recursive definitions:

Definition 5. *The set of* matching substitutions *of Definition 3 is redefined as:*

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

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

where H, G, S_h and $\mathbb{B}$ are as before in Definition 3, A is a conjunction of aggregates, and $\mathbb{S}$ is a set of identified CHR constraints (a CHR store).

A valid matching substitution now not only ensures that constraints S_h match the rule's head H, in a way satisfying its guard G, but also that the aggregate condition is satisfied for the aggregates A:

Definition 6 (Aggregate Condition). *For an aggregate A of the form* `aggregate`(s, i, d, f, T, G, R)*, a CHR store $\mathbb{S}$ and a built-in store $\mathbb{B}$:*

$$\mathsf{agg_cond}(A, \mathbb{S}, \mathbb{B}) = \quad s(V_0) \wedge \bigwedge_{k=1}^{n} i(V_{k-1}, \theta_k(T), V_k) \;\wedge\; f(V_n, R)$$

where $V_0, \ldots, V_n$ are new variables and $\{\theta_1, \ldots, \theta_n\} = \bigcup_{H \subseteq \mathbb{S}} \mathsf{matchings}'(G, H, \mathbb{S}, \mathbb{B})$. We extend this condition to (empty) conjunctions in the obvious way:

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

The definition is unambiguous if the increment predicate i corresponds to a commutative and associative operation. Finally, we modify the **Apply** transition from Definition 4 to use the modified set of matching substitutions $\mathsf{matchings}'$:

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

The propagation history does not record aggregates, so a rule is never fired more than once with the same combination of constraints, even if an aggregate's value changes. More information regarding this choice can be found in [18].

4 Implementation

Section 4.1 shows our aggregate framework supports incremental aggregate computation, and why this is necessary to facilitate efficient implementations. After that, Section 4.2 very briefly presents our prototype implementation.

4.1 Incremental Aggregate Maintenance

Section 2.2 introduced a naive, on-demand aggregate computation strategy, where aggregates are recomputed from scratch each time they are required. Computing an aggregate that way has time complexity $\mathcal{O}(S + IG + M + F)$, where S, I, and F are the time complexities of the `Start`, `Inc`, and `Final` operations respectively, G is the number of matches of `Goal`, and M is the time needed to find these matches. Usually, S, I, and F are $\mathcal{O}(1)$ and M is $\mathcal{O}(G)$, so the total complexity of a single aggregate computation is $\mathcal{O}(G)$.

If G is large, or the aggregate has to be computed many times, then it is more efficient to maintain the aggregate value incrementally. After an aggregate's value is first computed, this value is stored and maintained dynamically from then on. 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 the maintained value is invalidated, and no longer maintained incrementally. Instead, the first time the aggregate's value is needed again, it is recomputed from scratch.

Maintaining aggregate values reduces aggregate computations to cheap constant time lookups. Constraint insertions and removals, however, become slightly more expensive. Sometimes the maintenance cost may outweigh any performance gains. The development of effective static or dynamic analyses or heuristics that determine which computation strategy to use is left as future work.

In database literature, aggregates 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, incrementally computing holistic aggregates such as `min` and `max` requires, in general, a larger than constant-size working value. Because the working value can be arbitrarily large, our approach supports all three aggregate classes.

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 is recomputed from scratch the next time it is needed. In 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).

4.2 Prototype 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 [21]. The preprocessor transforms a CHR program with aggregates into a CHR program without aggregates. The latter can be dealt with by our existing CHR compiler. The transformation introduces auxiliary constraint symbols and rules, and adds new heads, guards and body goals to existing rules. This process is similar to the common approach often used by CHR programmers (cf. Section 2.1). But 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]. For lack of space, we refer to [18, 19] for a complete description.

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.

5.1 Expressiveness Case Studies

Figures 1, 2, and 3 contain different versions of the DIJKSTRA-*, EULER-*, and SUDOKU-* programs: on the left are versions without aggregates, on the right are versions which use the aggregates `argmin` and `no`, `forall` and `nb`, and `takemin` and `nb`, respectively. In all cases, the programs that use aggregates are not only more concise, but also more readable and maintainable. 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 can halve the program size w.r.t. to any of these metrics. We have also included figures for the BANKING example of Section 2.1 and the HOPCROFT-* programs. The latter programs implement Hopcroft's $\mathcal{O}(n \log n)$ state-minimizing algorithm [8], but are too long to reproduce here; they are listed in [18] instead.

5.2 Performance Evaluation

Because an implementation of aggregates has to deal with all possible use patterns of aggregates while the original programs are manually specialized, we expect the programs using aggregates to be slower than the original programs. Still, our prototype implementation shows that the runtime complexity can be maintained, with an acceptable constant overhead. Figure 4 plots benchmark results for the different versions of the DIJKSTRA, EULER, and HOPCROFT programs. For SUDOKU we did not find a scalable benchmark. The DIJKSTRA-agg-naive program uses a naive implementation of `argmin`, which results in an $\mathcal{O}(n^2)$ time complexity. The `argmin` aggregate in the DIJKSTRA-agg program though uses a Fibonacci heap to maintain the minimum, resulting in the optimal $\mathcal{O}(n \log n)$ time complexity. It is about three times as slow than the manually specialized DIJKSTRA-orig program. For the EULER and HOPCROFT programs, the version without aggregates is only about 1.5 times faster than the versions with aggregates.

The efficient Fibonacci heap implementation of the `argmin` aggregate illustrates another advantage of language support for aggregates: the data structures required for efficient aggregate computation only have to be implemented once; end users no longer have to worry about this.

6 Related Work

Constructs related to aggregates are found in many languages. In this section we briefly discuss some of them, and compare them to our approach where appropriate.

```
:- 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. 1. 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. 2. 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. 3. 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%
DIJKSTRA (Fig.1)	6	9	386	5	6	281	17%	33%	27%
EULER (Fig.2)	6	8	338	3	2	131	50%	75%	61%
HOPCROFT [18]	18	23	928	16	18	753	11%	22%	19%
SUDOKU (Fig.3)	5	9	385	3	4	207	40%	56%	46%

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

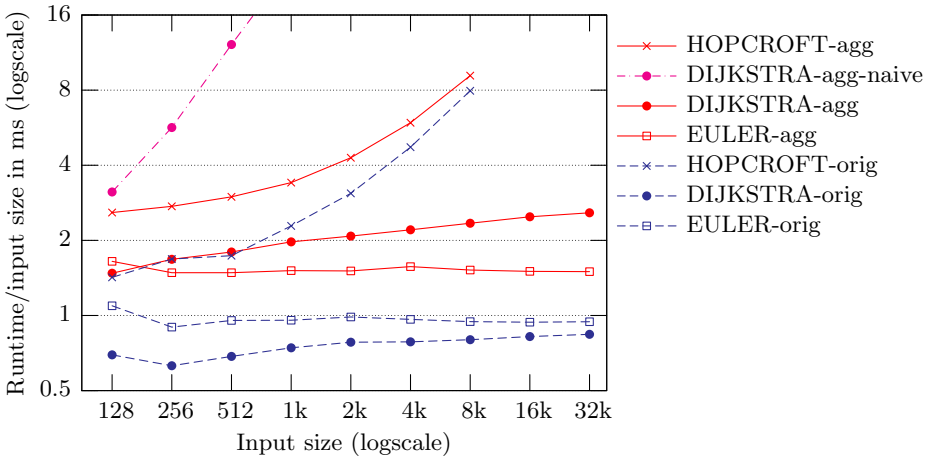

Fig. 4. Runtimes for three programs, with and without aggregates.

SQL. For SQL [2], the standard database query language, which unlike CHR [17] is not Turing-complete, aggregates are very important because they 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[4]. More recently, many database systems started to include the possibility to extend the database query language with user-defined aggregate functions.

Production Rule Systems. The first generation of production rule (PR) systems offered very limited support for aggregates: most systems had negation as absence (`\+`), some also included `exists` and `forall`. Very recently, some major PR systems [3, 5] introduced a general `accumulate` construct, similar to our `aggregate/7`. However, to the best of our knowledge, most if not all current PR system lack syntactic shorthands for commonly used aggregates (cf. Table 1), efficient incremental aggregate maintenance, and support for arbitrary nested aggregates or even conjunctions in the aggregate goal.

CHR$\overline{\neg}$. The `\+` aggregate — a shorthand for `count(G, 0)` — is similar to negation as absence in CHR$\overline{\neg}$ [20]: there are some minor differences, but CHR with aggregates can be considered as a generalization of CHR$\overline{\neg}$.

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

[4] e.g. `every` and `any/some` in SQL-99 and several statistical aggregates in SQL-2003

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[5] 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.

7 Conclusion and Future Work

In this paper we have proposed aggregates as a new language feature for CHR, a feature we believe will be welcomed by many practitioners. 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 includes a set of predefined aggregates, but also caters for user-defined application-specific aggregates. The generic aggregate operator is designed to facilitate efficient incremental computation. Benchmarks of our first implementation indicate that the desired complexity is attainable.

In future work, various optimizations and analyses can be researched to improve the efficiency of aggregates implementation. In particular, both specializations on the source level and dedicated support in the compiler can be considered. Incremental maintenance of aggregates can for instance be embedded directly in the constraint store insertion and removal operations. 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.

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 the anonymous referees of ICLP 2007 for their comments on an earlier version of this paper.

[5] See `http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx`.

References

1. ISO/IEC 13211:1995: Information technology – Programming languages – Prolog.
2. ISO/IEC 9075:2003: Information technology – Database languages – SQL.
3. JBoss Rules. http://www.jboss.com/products/rules.
4. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In *20th Intl. Conf. on Logic Programming*, LNCS 3132, Saint-Malo, France, 2004.
5. Ernest Friedman-Hill et al. The Jess Rule Engine. http://www.jessrules.com.
6. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
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 *Selected Contributions, First Workshop on Constraint Handling Rules*, May 2004. Home page at *http://www.cs.kuleuven.be/∼toms/CHR/*.
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 *Second Workshop on Constraint Handling Rules*, pages 3–17, Sitges, Spain, October 2005.
18. Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Bart Demoen. Aggregates in CHR. Technical Report CW481, Dept. Computer Science, K.U.Leuven, 2007.
19. Peter Van Weert, Jon Sneyers, and Bart Demoen. Aggregates for CHR through program transformation. In *17th Intl. Symposium on Logic-Based Program Synthesis and Transformation*, 2007. To appear.
20. Peter Van Weert, Jon Sneyers, Tom Schrijvers, and Bart Demoen. Extending CHR with negation as absence. In *Third Workshop on Constraint Handling Rules*, pages 125–139, Venice, Italy, 2006.
21. 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*.