The Correspondence Between the Logical Algorithms Language and CHR*

Leslie De Koninck**, Tom Schrijvers***, and Bart Demoen

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

Abstract This paper investigates the relationship between the Logical Algorithms language (LA) of Ganzinger and McAllester and Constraint Handling Rules (CHR). We present a translation scheme from LA to CHR^{rp}: CHR with rule priorities and show that the meta-complexity theorem for LA can be applied to a subset of CHR^{rp} via inverse translation. This result is compared with previous work. Inspired by the high-level implementation proposal of Ganzinger and McAllester, we demonstrate how LA programs can be compiled into CHR rules that interact with a scheduler written in CHR. This forms the first actual implementation of LA. Our implementation achieves the complexity required for the meta-complexity theorem to hold and can execute a subset of CHR^{rp} with strong complexity bounds.

1 Introduction

Constraint Handling Rules (CHR) [5] is a high-level rule based language, originally designed for the implementation of constraint solvers, but also more and more used as a general purpose programming language. Recently, it was shown that all algorithms can be implemented in CHR while preserving both time and space complexity [16]. We assume familiarity with CHR (see [5,13]).

In "Logical Algorithms" (LA) [9] (and based on previous work in [8,12]), Ganzinger and McAllester present a bottom-up logic programming language for the purpose of facilitating the derivation of complexity results of algorithms described by logical inference rules. This language resembles CHR in many ways and has often been referred to in the discussion of complexity results of CHR programs [1,6,15,17]. The aim of this paper is to investigate the relationship between both languages. More precisely, we look at how the meta-complexity theorem for LA can be applied to (a subset of) CHR, and how CHR can be used to implement LA.

First, we present a translation from LA to CHR^{rp}: CHR with rule priorities [10]. LA derivations of the original program correspond to CHR^{rp} derivations in

* A longer version of this paper is available as Technical Report [11].

** Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

*** Post-Doctoral Researcher of the Fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen).

the translation and vice versa. We show how to translate a subclass of CHR^{rp} to LA. This allows the meta-complexity theorem for LA to be applied to these CHR^{rp} programs as well. Because the LA meta-complexity theorem is based on an optimized implementation, it gives more accurate results than the implementation independent meta-complexity theorem of [7,6] while being more general than the ad-hoc complexity derivations in [15,17].

Our current prototype implementation of CHR^{rp} does not achieve the complexity required for the meta-complexity theorem to hold. Therefore, we propose an implementation of LA in (regular) CHR, which consists of the compilation of LA programs to CHR rules, combined with a scheduler written in CHR. By using a CHR implementation with advanced indexing support, such as the K.U.Leuven CHR system [14], our implementation achieves the required complexity. It is the first actual implementation of LA[1] and also a first implementation of a subset of CHR^{rp} with strong complexity bounds.

The rest of this paper is organized as follows. In Section 2, the syntax and semantics of the Logical Algorithms language and CHR^{rp} are reviewed. In Section 3 a translation of LA programs to CHR^{rp} programs is presented and in Section 4, the opposite is done for a subset of CHR^{rp}. The implementation of Logical Algorithms in refined operational semantics based CHR is given in Section 5. Section 6 shows that it has the required complexity. We conclude in Section 7.

2 Logical Algorithms and CHR^{rp}

In this section, we give an overview of the syntax and semantics of the Logical Algorithms language and CHR^{rp}.

2.1 Logical Algorithms

A Logical Algorithms program $P = \{r_1, \ldots, r_n\}$ is a set of rules. In [9], a graphical notation is used to represent rules. We use a textual representation that is closer to the syntax of CHR. A Logical Algorithms rule is an expression

$$r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$$

where r is the rule *name*, the atoms A_i (for $1 \leq i \leq n$) are the *antecedents* and C is the *conclusion*, which is a conjunction of atoms whose variables appear in the antecedents. Rule r has *priority* p where p is an arithmetic expression whose variables (if any) occur in the first antecedent A_1. If p contains variables, then r is called a dynamic priority rule. Otherwise, it is called a static priority rule.

The arguments of an atom are either Herbrand terms or (integer) arithmetic expressions. There are two types of atoms: comparisons and user-defined atoms. A comparison has the form $x < y$, $x \leq y$, $x = y$ or $x \neq y$ with x and y arithmetic expressions or, in case of $(=)/2$ and $(\neq)/2$, Herbrand terms. Comparisons are

[1] To the best of our knowledge and confirmed in personal communication with "Logical Algorithms" author David McAllester.

only allowed in the antecedents of a rule and all variables in a comparison must appear in earlier antecedents. A user-defined atom can be positive or negative. A negative user-defined atom has the form $del(A)$ where A is a positive user-defined atom. A ground user-defined atom is called an assertion.

Example 1. An example rule (from Dijkstra's shortest path algorithm as presented in [9]) with name `d2` and priority 1 is

```
d2 @ 1 : dist(V,D1), dist(V,D2), D2 < D1 => del(dist(V,D1)).
```

The antecedent `D2 < D1` is a comparison, the atoms `dist(V,D1)` and `dist(V,D2)` are positive user-defined antecedents. The negative ground atom `del(dist(a,5))` is an example of a negative assertion.

A Logical Algorithms state σ consists of a set of (positive and negative) assertions. Let $\mathcal{D}$ be the usual interpretation for the comparisons. Given a program P, the following transition converts one state into the next:

1. Apply $\sigma \overset{LA}{\longmapsto}_P \sigma \cup \theta(C)$ if there exists a (renamed apart) rule r in P of priority p of the form

$$r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$$

and a ground substitution θ such that for every antecedent A_i,
- $\mathcal{D} \models \theta(A_i)$ if A_i is a comparison
- $\theta(A_i) \in \sigma$ and $del(\theta(A_i)) \notin \sigma$ if A_i is a positive user-defined atom
- $\theta(A_i) \in \sigma$ if A_i is a negative user-defined atom

Furthermore, $\theta(C) \nsubseteq \sigma$ and no rule of priority p' and substitution θ' exists with $\theta'(p') < \theta(p)$ for which the above conditions hold.

A state is called final if no more transitions apply to it. A non-final state has priority p if the next firing rule instance has priority p. The condition $\theta(C) \nsubseteq \sigma$ ensures that no rule instance fires more than once and prevents trivial non-termination. Although the priorities restrict the possible derivations, the choice of which rule instance to fire from those with equal priority is non-deterministic.

A prefix instance of rule $r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$ is a tuple $\langle \theta(r), i \rangle$ with θ a ground substitution and $1 \leq i \leq n$. Its antecedents are $\theta(A_1), \ldots, \theta(A_i)$. The time complexity for running Logical Algorithms programs is given in [9] as $\mathcal{O}(|\sigma_0| + P_s + (P_d + A_d) \cdot \log N)$ where σ_0 is the initial state and $|\sigma_0|$ is its size. P_s is the number of *strong* prefix firings of static priority rules and P_d is the number of *strong* prefix firings of dynamic priority rules. A strong prefix firing is a prefix instance for which all antecedents hold in a state with priority lower or equal to the prefix' rule priority. A_d is the number of assertions that may participate in a dynamic priority rule instance. Finally, N is the number of distinct priorities.

2.2 CHR^{rp}: CHR with Rule Priorities

CHR^{rp} is CHR extended with user-definable rule priorities. It is introduced in [10] as a solution to the lack of high-level execution control in CHR. In CHR^{rp},

every rule is annotated with a rule priority that may depend on the arguments of the constraints in the rule heads, and a rule instance is only allowed to fire if no higher priority rule instance can. The operational semantics of CHR^{rp}, denoted by ω_p, is described as a state transition system.

As in ω_t, the theoretical operational semantics for CHR [2], we represent a state σ as a tuple $\langle G, S, B, T\rangle_n$, where G is the goal, a multiset of constraints; S is a set of identified CHR constraints, B is a conjunction of built-in constraints, T is the propagation history and n the next free identifier. The propagation history has a similar function as the $\theta(C) \notin \sigma$ condition in the LA semantics, but is less restrictive. The transitions of the ω_p semantics are shown below.[2]

1. **Solve** $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle G, S, c \wedge B, T\rangle_n$ where c is a built-in constraint.
2. **Introduce** $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle G, \{c\#n\} \cup S, B, T\rangle_{n+1}$ where c is a CHR constraint.
3. **Apply** $\langle \emptyset, H_1 \cup H_2 \cup S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle \theta(C), \theta(H_1 \cup S), \theta(B), T'\rangle_n$ where there exists a (renamed apart) rule in P of priority p of the form

$$r \ @ \ H_1' \backslash H_2' \iff g \mid C \text{ pragma priority}(p)$$

and a matching substitution θ such that $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$, $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$ and $t = id(H_1) +\!\!+ id(H_2) +\!\!+ [r] \notin T$. Furthermore, no rule of priority p' and substitution θ' exists with $\theta'(p') < \theta(p)$ for which the above conditions hold. $T' = T \cup \{t\}$.

The following theorem on the correspondence between the ω_p semantics of CHR^{rp} and the ω_t semantics of CHR, is proven in [10].

Theorem 1. *Every derivation D under ω_p is also a derivation under ω_t. If a state σ is a final state under ω_p, then it is also a final state under ω_t.*

CHR^{rp} differs from LA in the following ways. A Logical Algorithms state is a set of ground assertions. The CHR constraint store is a multi-set and may also contain non-ground constraints. In LA, built-in constraints are ask constraints and only include comparisons. CHR^{rp} supports any kind of built-in constraints. A removed CHR constraint may be reasserted and can then participate again in rule firings whereas a removed LA assertion cannot be asserted again. Finally, a LA rule may contain negated heads. In contrast, CHR^{rp} requires all heads to be positive.[3]

In the refined operational semantics of CHR [2], the textual order of the program rules determines which rule is tried next for the current *active* constraint. However, only rule instances in which the active constraint takes part are considered, and so a higher priority fireable rule instance in which the active

[2] We apply matching substitutions directly to the goal, CHR store and built-in store. If the rule bodies contain no built-in constraints, the built-in store remains empty.

[3] See [18] for an extension of CHR that allows negated heads.

constraint does not participate, will not fire. The textual rule order also does not support dynamic rule priorities.

3 Translating Logical Algorithms to CHR^{rp}

In this section, we show how Logical Algorithms can be translated into CHR^{rp} programs. CHR states of the translated program can be mapped on LA states of the original. With respect to this mapping, both programs have the same derivations.

3.1 The Translation Schema

The translation of a LA program P is denoted by $T(P) = T_{S/D}(P) \cup T_R(P)$. The contents of $T_{S/D}(P)$ and $T_R(P)$ are given below.

Set and Deletion Semantics We use an internal representation for assertions as CHR constraints consisting of the assertion itself and an extra argument, called the *mode indicator*, denoting whether it is positively asserted ("p"), negatively asserted ("n") or both ("b"). For every user-defined predicate a/n occurring in P, $T_{S/D}(P)$ contains the following rules:

$$\begin{aligned} a_r(\bar{X}, M) \setminus a(\bar{X}) &\iff M \neq \mathtt{n} \mid \mathit{true} \text{ pragma priority(1)} \\ a_r(\bar{X}, \mathtt{n}), a(\bar{X}) &\iff a_r(\bar{X}, \mathtt{b}) \text{ pragma priority(1)} \\ a(\bar{X}) &\iff a_r(\bar{X}, \mathtt{p}) \text{ pragma priority(2)} \end{aligned}$$

$$\begin{aligned} a_r(\bar{X}, M) \setminus \mathit{del}(a(\bar{X})) &\iff M \neq \mathtt{p} \mid \mathit{true} \text{ pragma priority(1)} \\ a_r(\bar{X}, \mathtt{p}), \mathit{del}(a(\bar{X})) &\iff a_r(\bar{X}, \mathtt{b}) \text{ pragma priority(1)} \\ \mathit{del}(a(\bar{X})) &\iff a_r(\bar{X}, \mathtt{n}) \text{ pragma priority(2)} \end{aligned}$$

If a representation already exists, one of the priority 1 rules updates this representation. Otherwise, one of the priority 2 rules generates a new representation. At lower priorities, it is guaranteed that every assertion, whether asserted positively, negatively or both, is represented by exactly one constraint in the store.[4]

Rules Given a LA rule $r \in P$ of the form

$$r \ @ \ p : A_1, \ldots, A_n \Rightarrow C$$

We first split up the antecedents into user-defined antecedents and comparison antecedents by using the *split* function defined below.

$$\mathit{split}([A|T]) = \begin{cases} \langle [A|A^u], A^c \rangle & \text{if } A \text{ is a user-defined atom} \\ \langle A^u, [A|A^c] \rangle & \text{if } A \text{ is a comparison} \end{cases}$$

$$\text{where } \mathit{split}(T) = \langle A^u, A^c \rangle$$

$$\mathit{split}([]) = \langle [], [] \rangle$$

[4] Under the ω_t semantics, this non-monotonic semantics cannot be ensured.

In the Logical Algorithms language, a given assertion may participate multiple times in the same rule instance, whereas in CHR all constraints in a single rule instance must be different. To overcome this semantic difference, a single LA rule is translated as a set of CHR rules such that every CHR rule covers a case of syntactically equal head constraints. Let $\langle A^u, A^c\rangle = \mathit{split}([A_1, \ldots, A_n])$ with $A^u = [A^u_1, \ldots, A^u_m]$ and $A^c = [A^c_1, \ldots, A^c_l]$. Let $\mathcal{P}$ be the set of all partitions of $\{1, \ldots, m\}$.[5] For a given partition $\rho \in \mathcal{P}$, the following function returns the most general unifier that unifies all antecedents $\{A_i \mid i \in S\}$ for every $S \in \rho$.

$$\mathit{partition_to_mgu}(\rho, [A^u_1, \ldots, A^u_m]) = \mathop{\circ}_{S\in\rho} \mathit{mgu}(\{A^u_i \mid i \in S\})$$

Let $\mathcal{PU} = \{\langle \rho, \theta\rangle \mid \rho \in \mathcal{P} \wedge \theta = \mathit{partition_to_mgu}(\rho, A^u) \wedge \mathcal{D} \models \bar{\exists}_\emptyset \theta(A^c)\}$. $\mathcal{PU}$ contains all partitions for which *partition_to_mgu* is defined and for which the comparison antecedents A^c are still satisfiable after applying the unifier. The next step is to filter out antecedents so that every set in the partition has only one representative. This is done by computing $\mathit{filter}(A^u, \langle \rho, \theta\rangle)$ for each $\langle \rho, \theta\rangle \in \mathcal{PU}$ where the *filter* function is as follows:

$$\mathit{filter}([A^u_i|T], \langle \rho, \theta\rangle) = \begin{cases} [\theta(A^u_i)|\mathit{filter}(T, \langle \rho, \theta\rangle)] & \text{if } \exists S \in \rho : i = \min(S) \\ \mathit{filter}(T, \langle \rho, \theta\rangle) & \text{otherwise} \end{cases}$$
$$filter([], _) = []$$

Finally, we add mode indicators to all remaining user-defined antecedents:

$$\mathit{modes}([A^{u'}|T]) = \begin{cases} \langle [a_r(\bar{X}, \mathtt{p})|A^m], N\rangle & \text{if } A^{u'} = a(\bar{X}) \\ \langle [a_r(\bar{X}, N')|A^m], [N' \neq \mathtt{p}|N]\rangle & \text{if } A^{u'} = \mathit{del}(a(\bar{X})) \end{cases}$$
$$\text{where } \langle A^m, N\rangle = \mathit{modes}(T)$$
$$\mathit{modes}([]) = \langle [], []\rangle$$

For every $\langle \rho, \theta\rangle \in \mathcal{PU}$, the CHR translation $T_R(P)$ contains a rule

$$r_\rho \ @ \ H \implies g_1, g_2 \mid C' \text{ pragma priority}(p + 2)$$

where $\langle H, g_1\rangle = \mathit{modes}(\mathit{filter}(A^u, \langle \rho, \theta\rangle))$, $g_2 = \theta(A^c)$ and $C' = \theta(C)$.

Example 2. A LA implementation of Dijkstra's shortest path algorithm is

```
d1 @ 1   : source(V) => dist(V,0).
d2 @ 1   : dist(V,D1), dist(V,D2), D2 < D1 => del(dist(V,D1)).
d3 @ D+2 : dist(V,D), e(V,C,U) => dist(U,D+C).
```

Its translation is

[5] $\mathcal{P}$ contains B_m elements in the worst case with B_m the mth Bell number.

```
e_r(V,C,U,M) \ e(V,C,U) <=> M \= n | true pragma priority(1).
e_r(V,C,U,n) , e(V,C,U) <=> e_r(V,C,U,b) pragma priority(1).
               e(V,C,U) <=> e_r(V,C,U,p) pragma priority(2).

e_r(V,C,U,M) \ del(e(V,C,U)) <=> M \= p | true pragma priority(1).
e_r(V,C,U,p) , del(e(V,C,U)) <=> e_r(V,C,U,b) pragma priority(1).
               del(e(V,C,U)) <=> e_r(V,C,U,n) pragma priority(2).

... % (similar rules for source/1 and dist/2)

d1_1   @ source_r(V,p) ==> dist(V,0) pragma priority(3).
d2_1/2 @ dist_r(V,D_1,p), dist_r(V,D_2,p) ==>
                          D_2 < D_1 | del(dist(V,D_1)) pragma priority(3).
d3_1/2 @ dist_r(V,D,p), e_r(V,C,U,p) ==> dist(U,D+C) pragma priority(D+4).
```

Example 3. A rule from the union-find implementation of [9] is the following:

```
uf4 @ 1 : union(X,Y), find(X,Z), find(Y,Z) => del(union(X,Y)).
```

Because antecedents `find(X,Z)` and `find(Y,Z)` are unifiable, this leads to the following two CHR rules:

```
uf4_1/2/3 @ union_r(X,Y,p), find_r(X,Z,p), find_r(Y,Z,p) ==>
                                    del(union(X,Y)) pragma priority(3).
uf4_1/23  @ union_r(X,X,p), find_r(X,Z,p) ==>
                                    del(union(X,X)) pragma priority(3).
```

3.2 The Correspondence Between LA and CHR^{rp} Derivations

In this section, we show that every derivation of the original program under the Logical Algorithms semantics, corresponds to a derivation of the translation under the ω_p semantics of CHR^{rp}. In order to do so, we introduce a mapping between (reachable) CHR execution states and LA states:

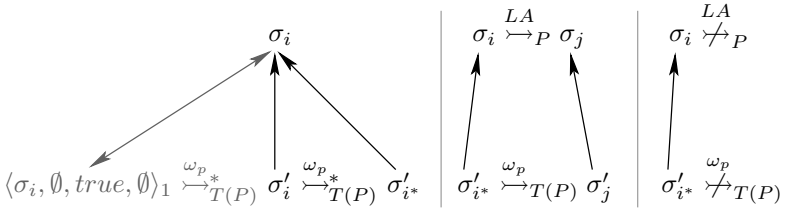
$$\begin{aligned} \mathit{chr_to_la}(\sigma) = &\{a(\bar{X}) \mid a(\bar{X}) \in A \vee (a_r(\bar{X}, M) \in A \wedge M \neq \mathtt{n})\} \\ &\cup \{\mathit{del}(a(\bar{X})) \mid \mathit{del}(a(\bar{X})) \in A \vee (a_r(\bar{X}, M) \in A \wedge M \neq \mathtt{p})\} \end{aligned}$$

where $\sigma = \langle G, S, B, T\rangle_n$ and $A = G \cup \mathit{chr}(S)$. The mapping function takes into account the constraints that are still in the goal or for which the set and deletion semantics rules have not fired yet.

Theorem 2. *For every reachable CHR^{rp} state σ, if $\sigma \overset{\omega_p}{\longmapsto}_{T(P)} \sigma'$ then either $\mathit{chr_to_la}(\sigma) = \mathit{chr_to_la}(\sigma')$ or $\mathit{chr_to_la}(\sigma) \overset{LA}{\longmapsto}_P \mathit{chr_to_la}(\sigma')$.*

Theorem 3. *For every Logical Algorithms state σ_i and reachable CHR^{rp} state σ'_i with $\mathit{chr_to_la}(\sigma'_i) = \sigma_i$, there exists a finite CHR^{rp} derivation $\sigma'_i \overset{\omega_p}{\longmapsto}^*_{T(P)} \sigma'_{i*}$ with $\mathit{chr_to_la}(\sigma'_{i*}) = \sigma_i$ such that if $\sigma_i \overset{LA}{\longmapsto}_P \sigma_j$ then $\sigma'_{i*} \overset{\omega_p}{\longmapsto}_{T(P)} \sigma'_j$ with $\mathit{chr_to_la}(\sigma'_j) = \sigma_j$ and if σ_i is a final state then σ'_{i*} is also a final state.*

Given a Logical Algorithms state σ, we can use $\langle \sigma, \emptyset, true, \emptyset \rangle_1$ as initial state for the CHR^{rp} derivation. If we extend the LA and CHR^{rp} transitions with appropriate labels, a LA program P and its translation $T(P)$ are weakly bisimilar.[6] Theorem 3 is illustrated in the figure below.



4 Translating CHR^{rp} Programs into Logical Algorithms

In the previous section, we have shown that Logical Algorithms can be translated into equivalent CHR^{rp} programs. In this section, we show how a particular subset of CHR^{rp} can be translated into equivalent Logical Algorithms programs. This allows us to apply the meta-complexity theorem for Logical Algorithms to these CHR^{rp} programs. The following three properties are required:

1. In all *reachable* states $\sigma = \langle G, S, B, T \rangle_n$: $vars(S) = \emptyset$.
2. All built-in constraints are comparisons; there are no built-in tell constraints.
3. A rule's priority depends on the arguments of at most *one* of its heads.

Here a state σ is reachable if there exists a derivation $\langle G, \emptyset, true, \emptyset \rangle_1 \overset{\omega_p}{\rightarrowtail}{}^*_P \sigma$. Because the order of heads in a CHR^{rp} rule is not important, we can assume without loss of generality that the priority depends on the left-most head only.[7] Given a program P satisfying these properties, a CHR^{rp} rule $r \in P$ of the form

$$r \ @ \ A_1, \ldots, A_m \backslash A_{m+1}, \ldots, A_n \iff g \mid C_1, \ldots, C_l \text{ pragma priority}(p)$$

is translated as

$$\begin{aligned} r \ @ \ p \ : \ & A_1^{id}, \ldots, A_n^{id}, \mathit{alldiff}(Id_1, \ldots, Id_n), g, \mathit{next_id}(Id_{next}) \Rightarrow \\ & \mathit{del}(A_{m+1}^{id}), \ldots, \mathit{del}(A_n^{id}), \mathit{del}(\mathit{next_id}(Id_{next})), \\ & C_1^{id}, \ldots, C_l^{id}, \mathit{next_id}(Id_{next} + l) \end{aligned}$$

where $A_i^{id} = \texttt{a(}\bar{X}\texttt{,}Id_i\texttt{)}$ if $A_i = \texttt{a(}\bar{X}\texttt{)}$, $C_i^{id} = \texttt{a(}\bar{X}\texttt{,}Id_{next} + i - 1\texttt{)}$ if $C_i = \texttt{a(}\bar{X}\texttt{)}$ and $\mathit{alldiff}(S) = \{(x \neq y) \mid x, y \in S \wedge x \neq y\}$. The initial database consists of the goal (where each constraint is extended with a unique identifier) and a $\texttt{next_id(}Id_{next}\texttt{)}$ assertion (with Id_{next} the next free identifier).

[6] A LA transition $\sigma \overset{LA}{\rightarrowtail}_P \sigma'$ is labeled $\sigma' \setminus \sigma$, a CHR^{rp} transition $\sigma \overset{\omega_p}{\rightarrowtail}_{T(P)} \sigma'$ is labeled $\mathit{chr_to_la}(\sigma') \setminus \mathit{chr_to_la}(\sigma)$ if this set is not empty, and τ (internal action) otherwise.

[7] We make abstraction of the syntactical limitations of simpagation rules.

Example 4. The following CHR^{rp} program implements a merge sort algorithm. Its input consists of a series of n (a power of 2) `number`/1 constraints. Its output is a sorted list of the numbers in the input, represented as `arrow`/2 constraints, where `arrow(`X`,`Y`)` indicates that X is right before Y.

```
ms1 @ arrow(X,A) \ arrow(X,B) <=> A < B | arrow(A,B) pragma priority(1).
ms2 @ merge(N,A),  merge(N,B) <=> A < B |
                          merge(2*N+1,A), arrow(A,B) pragma priority(2).
ms3 @ number(X) <=> merge(0,X) pragma priority(3).
```

Its Logical Algorithms translation is

```
ms1 @ 1 : arrow(X,A,Id1), arrow(X,B,Id2), A < B, next_id(NId) =>
                del(arrow(X,B,Id2)), arrow(A,B,NId),
                del(next_id(NId)), next_id(NId+1).
ms2 @ 2 : merge(N,A,Id1), merge(N,B,Id2), A < B, next_id(NId) =>
                del(merge(N,A,Id1)), del(merge(N,B,Id2)),
                merge(2*N+1,A,NId), arrow(A,B,NId+1),
                del(next_id(NId)), next_id(NId+2).
ms3 @ 3 : number(X,Id), next_id(NId) => del(number(X,Id)),
                merge(0,X,NId), del(next_id(NId)), next_id(NId+1).
```

Note that since the guard prevents the head constraints from being equal, the all different constraint on the constraint identifiers is not needed.

For this LA program, we can derive that given n initial `number`/2 assertions, there are $\mathcal{O}(n \log n)$ strong prefix firings and so using the meta-complexity theorem, we derive that the total runtime is $\mathcal{O}(n \log n)$. In contrast, when applying the meta-complexity theorem for CHR of [6], we find a runtime upperbound of $\mathcal{O}(n^3 \log n)$. The LA theorem clearly gives considerably more accurate results.

5 The Implementation of LA in CHR under ω_r

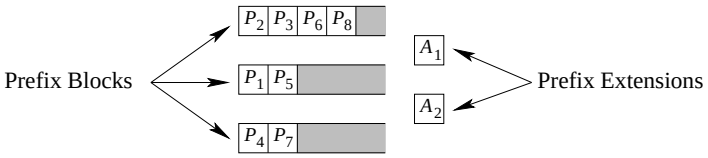
In this section, we present an implementation for LA in CHR under the refined operational semantics [2]. This implementation consists of the compilation of LA programs to CHR rules, combined with a scheduler module that is responsible for the execution control. It is based on the high-level implementation proposal of [9]. By using a CHR compiler with advanced indexing support, our implementation achieves the complexity required for the meta-complexity theorem of LA to hold. We make use of Prolog as host language, but the implementation can easily be adapted to work with other host languages.

5.1 Overview

The implementation is based on a form of eager matching, similar to the RETE algorithm [3]. Partial matches (called prefix instances in [9]) are stored and extended by new assertions. Full matches (rule instances) are inserted in a global

priority queue and the highest priority rule instance is fired. Only the highest priority partial matches are extended. This is enforced by storing partial matches in priority queues as well. Every partial match knows its priority as it is a rule instance prefix, and thus contains the leftmost head.

Every new assertion is scheduled to be combined with the highest priority partial match with which it has not been combined yet. After combining, it is rescheduled to be combined with the next partial match. This is done by using a data structure consisting of a series of *prefix blocks*: priority queues containing partial matches, alternated with a series of *prefix extensions*: the assertions with which the prefixes need to be combined. This is illustrated in the figure below. The highest priority elements in the data structure are scheduled for combination on the global priority queue. In the figure, these are the combination of prefix P_2 with extension A_1 and that of prefix P_1 with extension A_2. Full matches (rule instances) are scheduled on the global priority queue directly.



If a prefix and its extension share arguments, there is one data structure for each combination of these arguments, so that only feasible prefix/extension combinations are scheduled. For example, in the shortest path algorithm shown in the examples so far, there is a prefix/extension data structure for each different node, both for rule `d2` and rule `d3`.

5.2 The Compilation of LA Programs to CHR Rules

We present the compilation of Logical Algorithms to CHR rules by example. As example program, we use Dijkstra's shortest path algorithm. A compiled LA program consists of rules for

- Maintaining a representation for the assertions
- Removing the representation of invalidated prefix instances, prefix extensions and rule instances
- Generating and scheduling new such representations after a new assertion
- Extending prefix instances and firing rule instances

These rules are now described in more detail.

Representation of the Assertions For every positive user-defined atom $A = a(\bar{X})$ such that A is asserted or $del(A)$ is asserted, there exists a unique constraint representation $a_r(\bar{X}, M)$ where M is

- an uninstantiated Prolog variable if A is asserted and $del(A)$ is not
- the atom "`n`" if $del(A)$ is asserted and A is not

– the atom "b" if both A and $del(A)$ are asserted

In contrast with the representation used in Section 3.1, we use a Prolog variable as mode indicator for (strictly) positive assertions. Further on we show how this simplifies the task of removing invalidated prefix instances, prefix extensions and rule instances after a negative assertion.

Handling New Assertions For every new assertion, a constraint representation is created and merged with the existing representation if such exists. For a new positive or negative `dist`/2 assertion, this looks as follows:

```
    dist(V,D)  <=> dist_r(V,D,_).
del(dist(V,D)) <=> dist_r(V,D,n).

dist_r(V,D,M) \ dist_r(V,D,n) <=> var(M) | M = b.
dist_r(V,D,_) \ dist_r(V,D,_) <=> true.
```

Here, `var`/1 succeeds if its argument is an uninstantiated variable. The first two rules create a new representation. The third and fourth rule merge this representation with the existing representation if such exists. We rely on the refined operational semantics which states that rules are tried in textual order and occurrences are tried from right to left (so that the fourth rule always removes the most recently created representation).

Clean-up Prefix instances, prefix extensions and rule instances are all represented as constraints, $\alpha - \beta$ constraints for short using RETE terminology. The $\alpha - \beta$ constraints contain all arguments of their constituent antecedents that appear in the remaining antecedents or in the conclusion. They also contain the mode indicators of the representations of their positive constituent antecedents. Every $\alpha - \beta$ constraint has a unique identifier argument.

If A is a positive assertion, then a negative assertion $del(A)$ causes the mode indicator for A to be instantiated to the atom "b". This triggers all $\alpha - \beta$ constraints in which A occurs as a positive antecedent, which are then removed by rules like the following.

```
d1_ri(_,b,Id)     <=> remove_ri(Id).
d2_pi_1(_,_,b,Id) <=> remove_pi(Id).
d2_pe_1(_,b,Id)   <=> remove_pe(Id).
```

Here, `d1_ri`/3 represents a rule instance of rule `d1`, `d2_pi_1`/4 represents the first prefix instance of rule `d2` and `d2_pe_1`/3 represents its prefix extension. Similar clauses for rule `d3` exist. The calls to `remove_ri`/1, `remove_pi`/1 and `remove_pe`/1 remove respectively the rule instance, prefix instance and prefix extension from the scheduling data structures.

Scheduling New assertions are scheduled as rule instance (for `source`/1 in rule `d1`), as prefix instance (for `dist`/2 as first antecedent in rules `d2` and `d3`) or as prefix extension (for `dist`/2 as second antecedent in rule `d2` and `e`/3 as second antecedent in rule `d3`). For prefixes instances and extensions, the matching prefix/extension data structure is found by matching on a key consisting of the rule name, prefix number and arguments shared between prefix and extension.

```
source_r(V,M) ==> var(M) | d1_ri(V,M,Id_1), schedule_ri(1,Id_1).
dist_r(V,D,M) ==> var(M) | d2_pi_1(V,D,M,Id_1), schedule_pi(d2_1(V),1,Id_1),
                           d2_pe_1(D,M,Id_2), schedule_pe(d2_1(V),Id_2),
                           d3_pi_1(V,D,M,Id_3), schedule_pi(d3_1(V),D+2,Id_3).
e_r(V,C,U,M)  ==> var(M) | d3_pe_1(C,U,M,Id_1), schedule_pe(d3_1(V),Id_1).
```

We use fresh variables as identifiers for the generated $\alpha - \beta$ constraints. Predicate `schedule_ri`/2 schedules a rule instance with given priority and identifier, `schedule_pi`/3 schedules a prefix instance with given matching key (for finding the correct prefix/extension data structure), priority and identifier, and `schedule_pe`/2 schedules a prefix extension with given key and identifier.

Matching and Firing A rule instance is fired by asserting a `fire`/1 constraint which contains the identifier argument of its $\alpha - \beta$ constraint representation. A prefix instance is combined with a prefix extension by the assertion of a `cmb`/2 constraint which contains the identifiers of their $\alpha - \beta$ constraints.

```
d1_ri(V,_,I), fire(I) <=> dist(V,0).
d2_pi_1(V,D_1,_,I_1), d2_pe_1(D_2,_,I_2) \ cmb(I_1,I_2) <=> D_2 < D_1 | del(dist(V,D_1)).
d3_pi_1(V,D,_,I_1), d3_pe_1(C,U,_,I_2) \ cmb(I_1,I_2) <=> dist(U,D+C).
```

If the combination of a prefix instance with a prefix extension is another prefix instance, a new $\alpha - \beta$ constraint representation is made which is then scheduled for combination. Otherwise, the body of the combination rule corresponds to the conclusion of the rule that it implements.

5.3 Priority Queues

A priority queue or heap is a data structure that contains a set of prioritized items and supports the following operations: inserting and removing an item, finding a highest priority item and merging with another queue. The implementation proposal in [9] suggests the use of two types of priority queues, one for the fixed priorities, where each of the supported operations takes constant time, and a Fibonacci heap for the dynamic priorities.

Fibonacci heaps [4] are a type of priority queue that offer $\mathcal{O}(1)$ amortized time insertion, heap merging and finding a highest priority item, and $\mathcal{O}(\log n)$ amortized time item removal with n the number of items in the queue. It is suggested in [9] that by using only one node per priority, using linked lists to represent the items that share this priority, the item removal cost can be reduced

to $\mathcal{O}(\log N)$ with N the number of distinct priorities. This increases the cost of heap merging from $\mathcal{O}(1)$ for a single merge operation to a total cost of $\mathcal{O}(n \log N)$ for merging heaps when there are n items in total and N distinct priorities. Fortunately, this increased complexity does not influence the total complexity given by the meta-complexity theorem for Logical Algorithms.

The CHR implementation of the Fibonacci heaps is based on the description in [17] and extended to allow multiple heaps that can be merged and to use only one node for each distinct priority per heap.

5.4 The Scheduler

The scheduler implements the predicates `schedule_ri`/2, `schedule_pi`/3 and `schedule_pe`/2. It maintains the prefix/extension data structures and makes use of the priority queue implementations for this purpose. The scheduler initiates the firing of rule instances (by asserting a `fire`/1 constraint) and the combination of a prefix and extension (by asserting a `cmb`/2 constraint).

6 A Complexity Result

In Section 2.1, we have given the time complexity of Logical Algorithms. Theorem 4 shows that our implementation has the required complexity to make this result valid. Empirical evidence has confirmed this.

Theorem 4. *The time complexity of LA programs executed using our implementation is* $\mathcal{O}(|S_0| + P_s + (P_d + A_d) \cdot \log N)$ *with* S_0, P_s, P_d, A_d *and* N *as defined in Section 2.1.*

Proof (Sketch). The proof for the high-level implementation description in [9] can be used if the following holds:

- Inserting an element in a priority queue takes $\mathcal{O}(1)$ time. Deleting an element from one takes $\mathcal{O}(1)$ time for elements with a static priority and $\mathcal{O}(\log N)$ (amortized) time for elements with a dynamic priority. Merging two priority queues takes $\mathcal{O}(1)$ (amortized) time.
- Finding the first prefix block of a prefix/extension data structure for a given key consisting of a rule name, prefix number and the arguments shared between prefix and extension, takes constant time. The same holds for creating such a data structure should it not exist, and for adding a new prefix block at the end of such a structure.
- Finding a prefix instance, prefix extension or rule instance given its identifier takes constant time.

By using the advanced indexing supplied by the CHR compiler, our implementation satisfies these requirements, except that merging takes more than $\mathcal{O}(1)$ time. The heaps that are merged (when deleting a prefix extension) contain together up to P_d items.[8] As a result, the total cost of heap merging is $\mathcal{O}(P_d \cdot \log N)$.

[8] Although a prefix instance can "move" from one heap to another, each move operation corresponds to a (larger) prefix or rule instance.

For merging local priority queues, both for static and dynamic priorities, we need an optimal implementation of the union find algorithm. This supports quasi-constant lookup via, and unification of priority queue identifiers. Such an optimal implementation exists for CHR [15]. □

Although the cost of heap merging when using only one node for each distinct priority, appears to be larger than assumed in [9], the meta-complexity theorem remains valid. While the complexity requirements are very stringent, our high-level implementation in CHR is able to satisfy them.

7 Conclusions

In this paper, we have investigated the relationship between the Logical Algorithms language and Constraint Handling Rules. We have presented an elegant translation from LA to CHR^{rp}: CHR with rule priorities. The original program and its translation are essentially weakly bisimilar. A translation scheme is given that allows a subclass of CHR^{rp} to be translated into Logical Algorithms. This allows direct application of the meta-complexity theorem for Logical Algorithms to (the translation of) these CHR^{rp} programs, which gives more accurate results than the meta-complexity theorem for CHR given in [6].

By compiling LA rules into CHR rules and using a scheduler (also written in CHR) to control the execution, we are able to execute LA programs in any CHR implementation based on the refined operational semantics of CHR. We also achieve the required complexity by using the K.U.Leuven CHR system which uses optimized indexing structures. This result illustrates the strength of CHR for implementing complex systems in a concise way with optimal complexity. Moreover, when combining the CHR^{rp} to Logical Algorithms translation with the Logical Algorithms implementation in CHR, we have a first optimized implementation for a subset of CHR^{rp}.

Related Work In [7,6], Frühwirth presents a meta-complexity theorem for CHR programs containing only simplification rules. It uses level mappings to find an upperbound on the number of rule firings and makes a (highly pessimistic) worst case estimate of the time spent on trying rules in each derivation step. The approach is more or less independent of the actual CHR implementation used and so it often largely overestimates the actual time complexity. Tight complexity results have been derived for particular programs using ad hoc techniques in [15,17].

Future Work The Logical Algorithms approach applied to CHR^{rp} (Section 4), could be extended to a larger subclass of CHR^{rp} by allowing non-ground constraints and built-in (tell) constraints. This will require a more complex scheduler and it remains unclear what the effects will be for the complexity theorem. The derivation length of a program is bounded by monotone information growth in Logical Algorithms and by using level mapping in the work of Frühwirth. We plan to investigate the advantages and disadvantages of both approaches.

References

1. H. Christiansen. A constraint-based bottom-up counterpart to definite clause grammars. In *Recent Advances in Natural Language Processing III, Selected Papers*, volume 260 of *Current Issues in Linguistic Theory*, 2004.
2. G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. In *20th Intl. Conf. on Logic Programming*, volume 3132 of *LNCS*, pages 90–104, 2004.
3. C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artif. Intell.*, 19(1):17–37, 1982.
4. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.
5. T. Frühwirth. Theory and practice of constraint handling rules. *J. Log. Program.*, 37(1-3):95–138, 1998.
6. T. Frühwirth. As time goes by II: More automatic complexity analysis of concurrent rule programs. *Electr. Notes Theor. Comput. Sci.*, 59(3), 2001.
7. T. Frühwirth. As time goes by: Automatic complexity analysis of concurrent rule programs. In *8th Intl. Conf. Principles of Knowledge Representation and Reasoning*, pages 547–557, 2002.
8. H. Ganzinger and D. A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In *1st Intl. Joint Conf. on Automated Reasoning*, volume 2083 of *LNCS*, pages 514–528, 2001.
9. H. Ganzinger and D. A. McAllester. Logical algorithms. In *18th Intl. Conf. on Logic Programming*, volume 2401 of *LNCS*, pages 209–223, 2002.
10. L. De Koninck, T. Schrijvers, and B. Demoen. CHR^{rp}: Constraint handling rules with rule priorities. Technical Report CW 479, K.U.Leuven, Belgium, March 2007.
11. L. De Koninck, T. Schrijvers, and B. Demoen. The correspondence between the logical algorithms language and CHR. Technical Report CW 480, K.U.Leuven, Belgium, March 2007.
12. D. A. McAllester. On the complexity analysis of static analyses. In *6th Intl. Symp. on Static Analysis*, volume 1694 of *LNCS*, pages 312–329, 1999.
13. T. Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, 2005.
14. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: implementation and application. In *First Workshop on CHR: Selected Contributions*, volume 2004-01 of *Ulmer Informatik-Berichte*, pages 1–5. Universität Ulm, 2004.
15. T. Schrijvers and T. Frühwirth. Optimal union-find in constraint handling rules. *Theory and Practice of Logic Programming*, 6(1&2), 2006.
16. J. Sneyers, T. Schrijvers, and B. Demoen. The computational power and complexity of Constraint Handling Rules. In *2nd Workshop on Constraint Handling Rules*, volume 421 of *Reports CW*, pages 3–17. Dept. of Computer Science, K.U.Leuven, Belgium, 2005.
17. J. Sneyers, T. Schrijvers, and B. Demoen. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In *20th Workshop on Logic Programming*, volume 1843-06-02 of *INFSYS Research Report*, pages 182–191. TU Wien, 2006.
18. P. Van Weert, J. Sneyers, T. Schrijvers, and B. Demoen. Extending CHR with negation as absence. In *3rd Workshop on CHR*, volume 452 of *Reports CW*, pages 125–140. Dept. of Computer Science, K.U.Leuven, Belgium, 2006.