

# The Correspondence Between the Logical Algorithms Language and CHR

*Leslie De Koninck      Tom Schrijvers*  
*Bart Demoen*

*Report CW480, March 2007*



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

# The Correspondence Between the Logical Algorithms Language and CHR

*Leslie De Koninck      Tom Schrijvers*  
*Bart Demoen*

*Report CW480, March 2007*

Department of Computer Science, K.U.Leuven

## Abstract

This paper investigates the relationship between the Logical Algorithms formalism (LA) of Ganzinger and McAllester and Constraint Handling Rules (CHR). We present a translation scheme from LA to CHR<sup>rp</sup>: CHR with rule priorities and show that the meta-complexity theorem for LA can be applied to a subset of CHR<sup>rp</sup> 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 required complexity for the meta-complexity theorem to hold and can execute a subset of CHR<sup>rp</sup> with strong complexity bounds.

**Keywords :** Constraint Handling Rules, Meta-Complexity Theorem, Logical Algorithms.

**CR Subject Classification :** D.1.6 [Programming Techniques] Logic Programming, D.3.2 [Programming Languages] Language Classifications — Constraint and logic languages, F.2.0 [Analysis of Algorithms and Problem Complexity] General

# 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  
{leslie,toms,bmd}@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  $\text{CHR}^{\text{rp}}$ : CHR with rule priorities and show that the meta-complexity theorem for LA can be applied to a subset of  $\text{CHR}^{\text{rp}}$  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  $\text{CHR}^{\text{rp}}$  with strong complexity bounds.

## 1 Introduction

Constraint Handling Rules (CHR) [6] 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 [15]. We assume familiarity with CHR (see [6,12]).

In “Logical Algorithms” (LA) [9] (and based on previous work in [8,11]), 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 [1,7,14,16]. 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  $\text{CHR}^{\text{rp}}$ : CHR with rule priorities [10]. LA derivations of the original program correspond to  $\text{CHR}^{\text{rp}}$  derivations in

---

\* 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  $\text{CHR}^{\text{FP}}$  to LA. This allows the meta-complexity theorem for LA to be applied to these  $\text{CHR}^{\text{FP}}$  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 [5,7] while being more general than the ad-hoc complexity derivations in [14,16].

Our current prototype implementation of  $\text{CHR}^{\text{FP}}$  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 [13], our implementation achieves the required complexity. It is the first actual implementation of  $\text{LA}^1$  and also a first implementation of a subset of  $\text{CHR}^{\text{FP}}$  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  $\text{CHR}^{\text{FP}}$  are reviewed. In Section 3 a translation of LA programs to  $\text{CHR}^{\text{FP}}$  programs is presented and in Section 4, the opposite is done for a subset of  $\text{CHR}^{\text{FP}}$ . 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 $\text{CHR}^{\text{FP}}$

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

### 2.1 Logical Algorithms

A Logical Algorithms program  $P = \{r_1, \dots, r_n\}$  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

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

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

---

<sup>1</sup> 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  $D_2 < D_1$  is a comparison, the atoms  $dist(V, D_1)$  and  $dist(V, D_2)$  are positive user-defined antecedents. The negative ground atom  $del(dist(a, 5))$  is an example of a negative assertion.

A Logical Algorithms state  $\sigma$  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**  $\sigma \xrightarrow{LA} \sigma \cup \theta(C)$  if there exists a (renamed apart) rule  $r$  in  $P$  of priority  $p$  of the form

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

and a ground substitution  $\theta$  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,  $\theta(C) \notin \sigma$  and no rule of priority  $p'$  and substitution  $\theta'$  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  $\theta(C) \notin \sigma$  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  $r @ p : A_1, \dots, A_n \Rightarrow C$  is a tuple  $\langle \theta(r), i \rangle$  with  $\theta$  a ground substitution and  $1 \leq i \leq n$ . Its antecedents are  $\theta(A_1), \dots, \theta(A_i)$ . 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  $\sigma_0$  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<sup>FP</sup>: CHR with Rule Priorities

CHR<sup>FP</sup> 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<sup>FP</sup>,

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  $\text{CHR}^{\text{p}}$ , denoted by  $\omega_p$ , is described as a state transition system.

As in  $\omega_t$ , the theoretical operational semantics for CHR [2], we represent a state  $\sigma$  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  $\theta(C) \not\subseteq \sigma$  condition in the LA semantics, but is less restrictive. The transitions of the  $\omega_p$  semantics are shown below.<sup>2</sup>

- |  |
|--|
| <ol style="list-style-type: none"> <li>1. <b>Solve</b> <math>\langle \{c\} \uplus G, S, B, T \rangle_n \xrightarrow{\omega_p} \langle G, S, c \wedge B, T \rangle_n</math> where <math>c</math> is a built-in constraint.</li> <li>2. <b>Introduce</b> <math>\langle \{c\} \uplus G, S, B, T \rangle_n \xrightarrow{\omega_p} \langle G, \{c\#n\} \cup S, B, T \rangle_{n+1}</math> where <math>c</math> is a CHR constraint.</li> <li>3. <b>Apply</b> <math>\langle \emptyset, H_1 \cup H_2 \cup S, B, T \rangle_n \xrightarrow{\omega_p} \langle \theta(C), \theta(H_1 \cup S), \theta(B), T' \rangle_n</math> where there exists a (renamed apart) rule in <math>P</math> of priority <math>p</math> of the form</li> </ol> |
|--|

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

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

The following theorem on the correspondence between the  $\omega_p$  semantics of  $\text{CHR}^{\text{p}}$  and the  $\omega_t$  semantics of CHR, is proven in [10].

**Theorem 1.** *Every derivation  $D$  under  $\omega_p$  is also a derivation under  $\omega_t$ . If a state  $\sigma$  is a final state under  $\omega_p$ , then it is also a final state under  $\omega_t$ .*

$\text{CHR}^{\text{p}}$  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.  $\text{CHR}^{\text{p}}$  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,  $\text{CHR}^{\text{p}}$  requires all heads to be positive.<sup>3</sup>

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

<sup>2</sup> 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.

<sup>3</sup> See [17] 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<sup>rp</sup>

In this section, we show how Logical Algorithms can be translated into CHR<sup>rp</sup> 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:

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

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.

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

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

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

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

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  $\langle A^u, A^c \rangle = \text{split}([A_1, \dots, A_n])$  with  $A^u = [A_1^u, \dots, A_m^u]$  and  $A^c = [A_1^c, \dots, A_l^c]$ . Let  $\mathcal{P}$  be the set of all partitions of  $\{1, \dots, m\}$ .<sup>4</sup> 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  $S \in \rho$ .

$$\text{partition\_to\_mgu}(\rho, [A_1^u, \dots, A_m^u]) = \underset{S \in \rho}{\circ} \text{mgu}(\{A_i^u \mid i \in S\})$$

Let  $\mathcal{PU} = \{\langle \rho, \theta \rangle \mid \rho \in \mathcal{P} \wedge \theta = \text{partition\_to\_mgu}(\rho, A^u) \wedge \mathcal{D} \models \exists_0 \theta(A^c)\}$ .  $\mathcal{PU}$  contains all partitions for which  $\text{partition\_to\_mgu}$  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  $\text{filter}(A^u, \langle \rho, \theta \rangle)$  for each  $\langle \rho, \theta \rangle \in \mathcal{PU}$  where the  $\text{filter}$  function is as follows:

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

$$\text{filter}([], \_) = []$$

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

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

where  $\langle A^m, N \rangle = \text{modes}(T)$

$$\text{modes}([]) = \langle [], [] \rangle$$

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  $\langle H, g_1 \rangle = \text{modes}(\text{filter}(A^u, \langle \rho, \theta \rangle))$ ,  $g_2 = \theta(A^c)$  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

<sup>4</sup>  $\mathcal{P}$  contains  $B_m$  elements in the worst case with  $B_m$  the  $m^{\text{th}}$  Bell number.

```

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

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

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

d11 @ sourcer(V,p) ==> dist(V,0) pragma priority(3).
d21/2 @ distr(V,D1,p), distr(V,D2,p) ==>
      D2 < D1 | del(dist(V,D1)) pragma priority(3).
d31/2 @ distr(V,D,p), er(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:

```

uf41/2/3 @ unionr(X,Y,p), findr(X,Z,p), findr(Y,Z,p) ==>
      del(union(X,Y)) pragma priority(3).
uf41/23 @ unionr(X,X,p), findr(X,Z,p) ==>
      del(union(X,X)) pragma priority(3).

```

### 3.2 The Correspondence Between LA and CHR<sup>TP</sup> 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  $\omega_p$  semantics of CHR<sup>TP</sup>. In order to do so, we introduce a mapping between (reachable) CHR execution states and LA states:

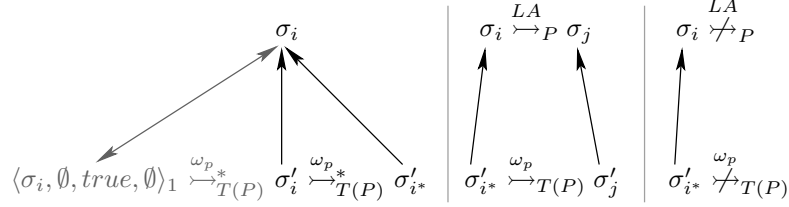
$$\begin{aligned}
chr\_to\_la(\sigma) = & \{a(\bar{X}) \mid a(\bar{X}) \in A \vee (a_r(\bar{X}, M) \in A \wedge M \neq n)\} \\
& \cup \{del(a(\bar{X})) \mid del(a(\bar{X})) \in A \vee (a_r(\bar{X}, M) \in A \wedge M \neq p)\}
\end{aligned}$$

where  $\sigma = \langle G, S, B, T \rangle_n$  and  $A = G \cup chr(S)$ . 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<sup>TP</sup> state  $\sigma$ , if  $\sigma \xrightarrow{\omega_p}_{T(P)} \sigma'$  then either  $chr\_to\_la(\sigma) = chr\_to\_la(\sigma')$  or  $chr\_to\_la(\sigma) \xrightarrow{LA}_P chr\_to\_la(\sigma')$ .*

**Theorem 3.** *For every Logical Algorithms state  $\sigma_i$  and reachable CHR<sup>TP</sup> state  $\sigma'_i$  with  $chr\_to\_la(\sigma'_i) = \sigma_i$ , there exists a finite CHR<sup>TP</sup> derivation  $\sigma'_i \xrightarrow{\omega_p^*}_{T(P)} \sigma'_{i^*}$  with  $chr\_to\_la(\sigma'_{i^*}) = \sigma_i$  such that if  $\sigma_i \xrightarrow{LA}_P \sigma_j$  then  $\sigma'_{i^*} \xrightarrow{\omega_p}_{T(P)} \sigma'_j$  with  $chr\_to\_la(\sigma'_j) = \sigma_j$  and if  $\sigma_i$  is a final state then  $\sigma'_{i^*}$  is also a final state.*

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



## 4 Translating $\text{CHR}^{\text{FP}}$ Programs into Logical Algorithms

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

1. In all *reachable* states  $\sigma = \langle G, S, B, T \rangle_n$ :  $\text{vars}(S) = \emptyset$ .
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  $\sigma$  is reachable if there exists a derivation  $\langle G, \emptyset, true, \emptyset \rangle_1 \xrightarrow{P}^* \sigma$ . Because the order of heads in a  $\text{CHR}^{\text{FP}}$  rule is not important, we can assume without loss of generality that the priority depends on the left-most head only.<sup>5</sup> Given a program  $P$  satisfying these properties, a  $\text{CHR}^{\text{FP}}$  rule  $r \in P$  of the form

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

is translated as

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

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

*Example 4.* The following  $\text{CHR}^{\text{FP}}$  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  $Y$ .

<sup>5</sup> We make abstraction of the syntactical limitations of simpagation rules.

```

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 [7], we find a runtime upperbound of  $\mathcal{O}(n^3 \log n)$ . The LA theorem clearly gives considerably more accurate results.

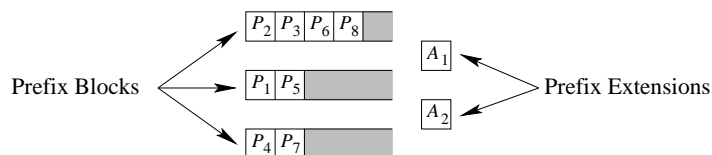
## 5 The Implementation of LA in CHR under $\omega_r$

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 Figure 1. The highest priority elements in the data structure are scheduled for combination on the global priority queue. For Figure 1, 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.



**Figure1.** Prefix Blocks and Extensions

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,n) <=> 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,Id1), schedule_ri(1,Id1).
dist_r(V,D,M) ==> var(M) | d2_pi_1(V,D,M,Id1), schedule_pi(d2_1(V),1,Id1),
                           d2_pe_1(D,M,Id2), schedule_pe(d2_1(V),Id2),
                           d3_pi_1(V,D,M,Id3), schedule_pi(d3_1(V),D+2,Id3).
e_r(V,C,U,M) ==> var(M) | d3_pe_1(C,U,M,Id1), schedule_pe(d3_1(V),Id1).

```

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,D1,_,I1), d2_pe_1(D2,_,I2) \ cmb(I1,I2) <=> D2 < D1 | del(dist(V,D1)).
d3_pi_1(V,D,_,I1), d3_pe_1(C,U,_,I2) \ cmb(I1,I2) <=> 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 [16] 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.<sup>6</sup> As a result, the total cost of heap merging is  $\mathcal{O}(P_d \cdot \log N)$ .

<sup>6</sup> 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 [14].  $\square$

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  $\text{CHR}^{\text{rp}}$ : CHR with rule priorities. The original program and its translation are essentially weakly bisimilar. A translation scheme is given that allows a subclass of  $\text{CHR}^{\text{rp}}$  to be translated into Logical Algorithms. This allows direct application of the meta-complexity theorem for Logical Algorithms to (the translation of) these  $\text{CHR}^{\text{rp}}$  programs, which gives more accurate results than the meta-complexity theorem for CHR given in [7].

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  $\text{CHR}^{\text{rp}}$  to Logical Algorithms translation with the Logical Algorithms implementation in CHR, we have a first optimized implementation for a subset of  $\text{CHR}^{\text{rp}}$ .

**Related Work** In [5,7], 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 [14,16].

**Future Work** The Logical Algorithms approach applied to  $\text{CHR}^{\text{rp}}$  (Section 4), could be extended to a larger subclass of  $\text{CHR}^{\text{rp}}$  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 *RANLP*, volume 260 of *Current Issues in Linguistic Theory*, 2003.
2. G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. In *ICLP'04: 20th International Conference 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. pages 324–341, 1990.
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. 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.
6. T. W. Frühwirth. Theory and practice of constraint handling rules. *J. Log. Program.*, 37(1-3):95–138, 1998.
7. T. W. Frühwirth. As time goes by II: More automatic complexity analysis of concurrent rule programs. *Electr. Notes Theor. Comput. Sci.*, 59(3), 2001.
8. H. Ganzinger and D. A. McAllester. A new meta-complexity theorem for bottom-up logic programs. In *IJCAR*, volume 2083 of *LNCS*, pages 514–528, 2001.
9. H. Ganzinger and D. A. McAllester. Logical algorithms. In *ICLP*, volume 2401 of *LNCS*, pages 209–223, 2002.
10. L. De Koninck, T. Schrijvers, and B. Demoen. CHR<sup>FP</sup>: Constraint handling rules with rule priorities. Technical Report CW 479, K.U.Leuven, Belgium, March 2007.
11. D. A. McAllester. On the complexity analysis of static analyses. In *SAS*, volume 1694 of *LNCS*, pages 312–329, 1999.
12. T. Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, Jun 2005.
13. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: implementation and application. In *First workshop on CHR: selected contributions*, pages 1–5, 2004.
14. T. Schrijvers and T. Frühwirth. Optimal union-find in constraint handling rules. *Theory and Practice of Logic Programming*, 6(1&2), 2006.
15. J. Sneyers, T. Schrijvers, and B. Demoen. The computational power and complexity of constraint handling rules. In *2nd Workshop on CHR*, 2005.
16. J. Sneyers, T. Schrijvers, and B. Demoen. Dijkstra’s algorithm with Fibonacci heaps: An executable description in CHR. In *WLP*, volume 1843-06-02 of *INFSYS Research Report*, pages 182–191. Technische Universität Wien, Austria, 2006.
17. P. Van Weert, J. Sneyers, T. Schrijvers, and B. Demoen. Extending CHR with negation as absence. In *3rd Workshop on CHR*, 2006.

## A Correspondence Proof

In this section, we prove the correspondence theorems from Section 3.2. In all what follows, reachability is considered with respect to initial states of the form  $\langle G, \emptyset, true, \emptyset \rangle_n$  where  $G$  is a set of constraints of the form  $a(\bar{X})$  and  $del(a(\bar{X}))$  and does not include constraints of the form  $a_r(\bar{X}, M)$  nor built-in constraints. Since no translation  $T(P)$  of a Logical Algorithms program  $P$  contains built-in tell constraints, the built-in store equals  $true$  in all reachable states. Another important invariant of reachable CHR execution states is given by the following lemma.

**Lemma 1.** *In every reachable state  $\sigma = \langle G, S, true, T \rangle_n$ , if  $a_r(\bar{X}, M_1) \# i_1 \in S$  and  $a_r(\bar{X}, M_2) \# i_2 \in S$  then  $M_1 = M_2$  and  $i_1 = i_2$ .*

*Proof.* The initial goal does not contain any constraints of the form  $a_r(\bar{X}, M)$ . The only rules that have a constraint  $a_r(\bar{X}, M)$  in their body are the following:

$$\begin{aligned} a_r(\bar{X}, \mathbf{n}), a(\bar{X}) &\iff a_r(\bar{X}, \mathbf{b}) \text{ pragma priority}(1) \\ a_r(\bar{X}, \mathbf{p}), del(a(\bar{X})) &\iff a_r(\bar{X}, \mathbf{b}) \text{ pragma priority}(1) \\ a(\bar{X}) &\iff a_r(\bar{X}, \mathbf{p}) \text{ pragma priority}(2) \\ del(a(\bar{X})) &\iff a_r(\bar{X}, \mathbf{n}) \text{ pragma priority}(2) \end{aligned}$$

The first two rules replace an existing constraint representation for respectively  $a(\bar{X})$  and  $del(a(\bar{X}))$ . The last two rules can only fire if no such representation exists, because otherwise one of the first two rules, or one of

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

would have fired, removing respectively the constraint  $a(\bar{X})$  or  $del(a(\bar{X}))$ . In either case, the invariant holds after firing these rules.  $\square$

The following lemma shows that every reachable state is *pre-normalized* before trying rules with priority  $> 2$ .

**Lemma 2.** *For every reachable state  $\sigma$ , there exists a finite derivation  $D = \sigma \xrightarrow{\omega_p}_{T(P)} \sigma^*$  with  $\sigma^* = \langle \emptyset, S, true, T \rangle_n$  such that all constraints in  $S$  are of the form  $a_r(\bar{X}, M) \# i$ ,  $chr\_to\_la(\sigma) = chr\_to\_la(\sigma^*)$  and all rules fired in  $D$  have priority 1 or 2. The state  $\sigma^*$  is in pre-normal form. Every state has a unique pre-normal form up to renaming of the CHR constraint identifiers.*

*Proof.* Let  $\|\sigma\| = 2 \cdot |\{a(\bar{X}) \mid a(\bar{X}) \in A\} \uplus \{del(a(\bar{X})) \mid del(a(\bar{X})) \in A\}| + |G|$  where  $\sigma = \langle G, S, true, T \rangle_n$ ,  $A = G \uplus chr(S)$  and if  $X$  is a (multi-)set,  $|X|$  is its cardinality. If  $\|\sigma\| = 0$  then  $\sigma$  is in pre-normal form. If  $\sigma$  is not in pre-normal form, then there exists at least one transition  $\sigma \xrightarrow{\omega_p}_{T(P)} \sigma'$ . For all such transitions  $chr\_to\_la(\sigma) = chr\_to\_la(\sigma')$  and  $\|\sigma'\| < \|\sigma\|$  (ensuring termination).

If the goal  $G$  is not empty, then only the **Introduce** transition is applicable. Every application of this transition decreases the size of  $G$  by one and moves a CHR constraint from the goal to the CHR constraint store, so  $\|\sigma'\| = \|\sigma\| - 1$ . If the goal  $G$  is empty then since  $\sigma$  is not in pre-normal form,  $chr(S)$  contains a constraint of the form  $a(\bar{X})$  or  $del(a(\bar{X}))$ . We look into detail to the case  $a(\bar{X}) \in chr(S)$ , the case  $del(a(\bar{X})) \in chr(S)$  is similar.

If  $a_r(\bar{X}, \mathbf{p}) \in chr(S)$  or  $a_r(\bar{X}, \mathbf{b}) \in chr(S)$  then the following rule of  $T(P)$  is applicable:

$$a_r(\bar{X}, M) \setminus a(\bar{X}) \iff M \neq \mathbf{n} \mid \text{true pragma priority}(1)$$

Firing the rule removes a constraint  $a(\bar{X})\#i$  from  $S$ , so  $\|\sigma'\| = \|\sigma\| - 2$ . If  $a_r(\bar{X}, \mathbf{n}) \in chr(S)$  then the following rule of  $T(P)$  is applicable:

$$a_r(\bar{X}, \mathbf{n}), a(\bar{X}) \iff a_r(\bar{X}, \mathbf{b}) \text{ pragma priority}(1)$$

Firing this rule removes a constraint  $a(\bar{X})\#i$  from  $S$  and adds a constraint  $a_r(\bar{X}, \mathbf{b})$  to  $G$ , so  $\|\sigma'\| = \|\sigma\| - 1$ . Finally, if no rule of priority 1 can be applied, which implies that no constraint of the form  $a_r(\bar{X}, M) \in chr(S)$ , then the following  $T(P)$  rule can fire:

$$a(\bar{X}) \iff a_r(\bar{X}, \mathbf{p}) \text{ pragma priority}(2)$$

This rule removes a constraint  $a(\bar{X})\#i$  from  $S$  and adds a constraint  $a_r(\bar{X}, \mathbf{p})$  to  $G$ , so again  $\|\sigma'\| = \|\sigma\| - 1$ .

In any case, if the goal is empty and  $\sigma$  is not in pre-normal form, a rule of priority 1 or 2 can fire and so no rule with lower priority is applicable.  $\square$

The state  $\sigma^*$  is called a pre-normalization of  $\sigma$ .

**Definition 1 (Implied Rule Instance).** A rule instance  $\theta(r)$  is implied in a state  $\sigma$  if  $\theta(C) \subseteq chr\_to\_la(\sigma)$  with  $\theta(C)$  the conclusion of  $\theta(r)$ .

**Lemma 3.** Given a pre-normalized state  $\sigma = \langle \emptyset, S, true, T \rangle_n$ . If there exists a transition  $\sigma \xrightarrow{\omega_p}_{T(P)} \sigma'$  in which an implied rule instance fires, then the pre-normalization of  $\sigma'$  has the form  $\langle \emptyset, S, true, T' \rangle_{n'}$ .

*Proof.* Let  $\sigma' = \langle G, S, true, T \rangle_n$ . Since  $chr\_to\_la(\sigma) = chr\_to\_la(\sigma')$ , it holds that if  $a(\bar{X}) \in G$  then  $a(\bar{X}, \mathbf{p}) \in chr(S)$  or  $a(\bar{X}, \mathbf{b}) \in chr(S)$  and if  $del(a(\bar{X})) \in G$  then  $a(\bar{X}, \mathbf{n}) \in chr(S)$  or  $a(\bar{X}, \mathbf{b}) \in chr(S)$ . Now all constraints in the goal are first introduced in the CHR constraint store and then they are removed one by one using one of the following rules:

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

These rules remove all the constraints that were introduced from the goal and do not change the rest of the CHR constraint store.  $\square$

Because the CHR constraint store remains unchanged after firing an implied rule instance and pre-normalizing the resulting state, only finitely many such rule instances can fire before either reaching a final execution state, or a state in which a non-implied rule instance can fire. We call such a state *normalized*.

**Definition 2 (Normal Form).** A pre-normalized CHR execution state  $\sigma$  is in normal form if it is a final state ( $\sigma \not\rightarrow_{T(P)}^{\omega_p}$ ) or there exists a transition  $\sigma \xrightarrow_{T(P)}^{\omega_p} \sigma'$  such that  $\text{chr\_to\_la}(\sigma) \subsetneq \text{chr\_to\_la}(\sigma')$ .

**Lemma 4.** For every Logical Algorithms state  $\sigma_{LA}$  and every normalized CHR execution state  $\sigma = \langle \emptyset, S, \text{true}, T \rangle_n$  such that  $\sigma_{LA} = \text{chr\_to\_la}(\sigma)$ , there exists a transition  $\sigma_{LA} \xrightarrow{LA} \sigma'_{LA}$  if and only if there exists a transition  $\sigma \xrightarrow_{T(P)}^{\omega_p} \sigma'$  firing a non-implied rule instance such that  $\sigma'_{LA} = \text{chr\_to\_la}(\sigma')$ .

*Proof.* A transition of  $\sigma_{LA}$  to  $\sigma'_{LA}$  implies there exists a fireable rule instance  $\theta(r)$  of a rule  $r$  in  $P$  with priority  $p$  of the form

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

Let  $\langle A^u, A^c \rangle = \langle [A_1^u, \dots, A_m^u], [A_1^c, \dots, A_l^c] \rangle = \text{split}([A_1, \dots, A_n])$  where we use the *split* function defined in Section 3.1. The user-defined antecedents can be partitioned into sets of syntactically equal antecedents with respect to the matching substitution  $\theta$ . The following function returns this partition:

$$\text{substitution\_to\_partition}(\theta, [A_1^u, \dots, A_m^u]) = \bigcup_i \{j \mid \theta(A_i^u) = \theta(A_j^u)\}$$

Let  $\rho = \text{substitution\_to\_partition}(\theta, A^u)$ . From the partition, we find the most general unifier  $\theta'$  that unifies all antecedents  $\{A_i^u \mid i \in S\}$  for every  $S \in \rho$ :  $\theta' = \text{partition\_to\_mgu}(\rho, A^u)$  with *partition\\_to\\_mgu* as defined in Section 3.1. Clearly,  $\theta'$  exists and is more general than  $\theta$ . For all comparison antecedents  $A_i^c$  with  $1 \leq i \leq l$ ,  $\mathcal{D} \models \theta(A_i^c)$ , so it also holds that  $\mathcal{D} \models \exists_{\emptyset} \theta'(A^c)$  and consequently a rule  $r_\rho$  exists. This rule looks as follows:

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

with  $\langle H, g_1 \rangle = \text{modes}(A^f)$ ,  $A^f = [A_1^f, \dots, A_k^f] = \text{filter}(A^u, \langle \rho, \theta' \rangle)$ ,  $g_2 = \theta'(A^c)$  and  $C' = \theta'(C)$ . The *modes* and *filter* functions are as defined in Section 3.1.

Let  $\theta''$  be a ground matching substitution such that  $\theta = \theta''|_{\text{vars}(\theta)} \circ \theta'$  where  $\theta''|_{\text{vars}(\theta)}$  is the projection of  $\theta''$  on the variables in  $\theta$ . Since  $\theta'$  is more general than  $\theta$ ,  $\theta''$  exists. For all  $i \in \{1, \dots, k\}$ , if  $A_i^f = a(\bar{X})$  then  $\theta''(a(\bar{X})) \in \sigma_{LA}$  and  $\theta''(\text{del}(a(\bar{X}))) \notin \sigma_{LA}$ , so  $H'_i = \theta''(a_r(\bar{X}, \mathbf{p})) \# \text{id}_i \in S$ . Furthermore  $H_i = a_r(\bar{X}, \mathbf{p})$  and so  $\theta''(H_i) = \text{chr}(H'_i)$ . If  $A_i^f = \text{del}(a(\bar{X}))$  then  $\theta''(\text{del}(a(\bar{X}))) \in \sigma_{LA}$ . As a result  $H'_i = \theta''(a_r(\bar{X}, N')) \# \text{id}_i \in S$  with  $N' = \mathbf{n}$  or  $N' = \mathbf{b}$ .  $H_i = a_r(\bar{X}, N)$  and  $g_1$  contains  $N \neq \mathbf{p}$ . By further imposing that  $\theta''(N) = N'$ ,  $\theta''(H_i) = \text{chr}(H'_i)$ .

All  $A_i^f$  are different for  $1 \leq i \leq k$ , and therefore, all  $id_i$  must be different. From  $\mathcal{D} \models \theta(A_i^c)$  for  $1 \leq i \leq l$  and because  $\theta''(g_1) = [N_1 \neq \mathbf{p}, \dots, N_o \neq \mathbf{p}]$  with  $N_j = \mathbf{n}$  or  $N_j = \mathbf{b}$  for  $1 \leq j \leq o$ ,  $\mathcal{D} \models \theta''(g_1 \wedge g_2)$ . We conclude that  $\theta''$  is a matching substitution that matches the head with constraints from  $S$  and causes the guard to be satisfied.

It is not possible that  $id(H) \text{ ++ } [r_\rho] \in T$  because  $chr\_to\_la$  grows monotonically, which implies that  $\theta(C) = \theta''(C') \in chr\_to\_la(\sigma) = \sigma_{LA}$  which contradicts with the applicability of  $\theta(r)$  in  $\sigma_{LA}$ .

If we ignore rule priorities, all conditions are satisfied so that rule instance  $\theta(r_\rho)$  can fire. The resulting state  $\sigma'$  has the form  $\langle \theta(C), S, true, T' \rangle_n$ . Clearly, if  $\sigma_{LA} = chr\_to\_la(\langle \emptyset, S, true, T \rangle_n)$  and  $\sigma'_{LA} = \sigma_{LA} \cup \theta(C)$  then  $\sigma'_{LA} = chr\_to\_la(\sigma')$ . We now prove that every CHR transition corresponds to a Logical Algorithms transition, also ignoring rule priorities. Both results combined give us that the priority of the highest priority rule instance is equal in both  $\sigma$  and  $\sigma_{LA}$ .

A transition of  $\sigma = \langle \emptyset, S, true, T \rangle_n$  to  $\sigma'$  implies that  $T(P)$  contains a rule

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

and so the Logical Algorithms program  $P$  contains a rule

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

Let  $\langle A^u, A^c \rangle = split([A_1, \dots, A_n])$  and  $\theta = partition\_to\_mgu(\rho, A^u)$ . If  $A_i = a(\bar{X}) \in A^u$  then  $\theta(a_r(\bar{X}, \mathbf{p})) \in H$ . If  $A_i = del(a(\bar{X})) \in A^u$  then  $\theta(a_r(\bar{X}, N)) \in H$  and  $(N \neq \mathbf{p}) \in g_1$ . Finally, if  $A_i \in A^c$  then  $\theta(A_i) \in g_2$ . There exists a (ground) matching substitution  $\theta'$  such that  $\theta'(H) \in chr(S)$  and  $\mathcal{D} \models \exists_0 \theta'(g_1 \wedge g_2)$ .

Let  $\theta'' = \theta' \circ \theta$  and let  $\sigma_{LA} = chr\_to\_la(\sigma)$ . Because  $\theta'$  is a ground substitution,  $\mathcal{D} \models \exists_0 \theta'(g_1 \wedge g_2)$  implies that for all  $A_i \in A^c$ ,  $\mathcal{D} \models \theta''(A_i)$ . For all positive user-defined antecedents  $A_i = a(\bar{X}) \in A^u$ , we have that  $\theta''(a(\bar{X}, \mathbf{p})) \in chr(S)$  and so  $\theta''(A_i) \in \sigma_{LA}$  and  $del(\theta''(A_i)) \notin \sigma_{LA}$ . For all negative user-defined antecedents  $A_i = del(a(\bar{X})) \in A^u$ , we have that  $\theta''(a_r(\bar{X}, N)) \in chr(S)$  with  $N = \mathbf{b}$  or  $N = \mathbf{n}$  and so  $\theta''(A_i) \in \sigma_{LA}$ . We have assumed that  $\theta'(r_\rho)$  is not an implied rule instance and so  $\theta'(C') = \theta''(C) \notin \sigma_{LA}$ .

If we again ignore rule priorities, all conditions are satisfied so that rule instance  $\theta''(r)$  can fire in state  $\sigma_{LA}$  and it holds that  $\sigma'_{LA} = \sigma_{LA} \cup \theta''(C) = chr\_to\_la(\langle \theta'(C'), S, true, T \rangle_n)$ . Now we have that both the original program  $P$  and its translation  $T(P)$  can fire corresponding rule instances if we ignore priorities, and so their highest priority rule instances also correspond.  $\square$

**Theorem 2.** For every reachable  $CHR^p$  state  $\sigma$ , if  $\sigma \xrightarrow{\omega_p}_{T(P)} \sigma'$  then either  $chr\_to\_la(\sigma) = chr\_to\_la(\sigma')$  or  $chr\_to\_la(\sigma) \xrightarrow{LA}_P chr\_to\_la(\sigma')$ .

*Proof.* Implied by Lemmas 2, 3 and 4.  $\square$

**Theorem 3.** For every Logical Algorithms state  $\sigma_i$  and reachable  $CHR^p$  state  $\sigma'_i$  with  $chr\_to\_la(\sigma'_i) = \sigma_i$ , there exists a finite  $CHR^p$  derivation  $\sigma'_i \xrightarrow{\omega_p}_{T(P)} \sigma'_{i^*}$  with  $chr\_to\_la(\sigma'_{i^*}) = \sigma_i$  such that if  $\sigma_i \xrightarrow{LA}_P \sigma_j$  then  $\sigma'_{i^*} \xrightarrow{\omega_p}_{T(P)} \sigma'_j$  with  $chr\_to\_la(\sigma'_j) = \sigma_j$  and if  $\sigma_i$  is a final state then  $\sigma'_{i^*}$  is also a final state.

*Proof.* Implied by Lemmas 2, 3 and 4. □

## B On The Complexity of Dijkstra's Shortest Path algorithm

Dijkstra's shortest path algorithm can be implemented in the Logical Algorithms language using the following three rules [9]:

```
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 complexity crucially depends on the number of strong prefix firings of rule **d3**. A **dist/2** assertion can only be added by rules **d1** and **d3**, and the number of **dist/2** assertions generated by rule **d1** is limited to one, because there is only one **source/1** assertion in the input. Since rule **d3** runs at a lower priority than rule **d2**, there can be no two consecutive **dist/2** assertions without rule **d2** being checked and potentially fired if there are two **dist/2** assertions for the same node. So, at any time, there can be at most  $n + 1$  positive **dist/2** assertions in the database with  $n$  the number of nodes.

This means that at any time, the items scheduled on the global priority queue have at most  $n + 2$  distinct priorities. Since the number of **dist/2** constraints for the same node is limited to at most two (which may happen after firing rule **d3** and before firing rule **d2**), the local priority queues in the prefix blocks for rule **d3** all contain no more than two elements, and removing an element from one of them takes constant time.

When rule **d3** fires at some priority  $p$ , it can only add **dist/2** assertions with a distance longer than  $p - 2$  and so it can never fire at a higher priority than  $p$  and all firings for the same priority are consecutive.

The global queue can be implemented as a Fibonacci heap with one node per priority class, which allows  $\mathcal{O}(\log n)$  item removal if there are  $n$  distinct priorities, and even  $\mathcal{O}(1)$  removal if there are other items with the same priority in the queue.

Since rule **d3** will fire at most  $e$  times with  $e$  the number of edges and at most  $\mathcal{O}(n)$  different priorities, and since all firings at the same priority are consecutive, the total complexity for rule **d3**, as well as for the entire algorithm, is  $\mathcal{O}(e + n \log n)$ , which is optimal.

## C Benchmarks

In this section, some benchmark results are presented as an empirical validation of Theorem 4. The benchmarks are performed on a Pentium IV 2.8 GHz using SWI-Prolog 5.6.0. The times reported are in seconds and do not include time spent on garbage collection.

### C.1 Union Find and Dijkstra’s Shortest Path

Table 1 shows runtimes for the union-find and shortest path algorithms presented in [9]. For union-find, we use a set of  $n$  randomly generated union assertions over  $n$  distinct items. The union-find algorithm has a theoretical worst case complexity of  $\mathcal{O}(n \log n)$ , but scales almost linearly in the example.<sup>7</sup> For Dijkstra’s shortest path algorithm, the input consists of  $n$  nodes, connected by  $4n$  edges. The theoretical worst case complexity according to the meta complexity theorem is  $\mathcal{O}(e \log n)$  with  $e$  the number of edges, or  $\mathcal{O}(n \log n)$  for the sparse graphs used in the benchmark.

$n$	Union-Find	Dijkstra
32	0.27	0.24
64	0.61	0.46
128	1.31	0.95
256	2.53	2.00
512	5.30	4.19
1024	10.61	9.05
2048	22.09	18.77

**Table 1.** Union-Find and Shortest Path

### C.2 Heap Sort

The LA program consisting of the following rule sorts integer numbers:

```

s1 @ I : integer(I), current(P) =>
        del(integer(I)), del(current(P)), position(P,I), current(P+1).
```

The input database consists of a series of `integer(I)` assertions, with  $I$  an integer, representing the numbers to be sorted. It also contains the assertion `current(1)` denoting the fact that the first unoccupied position is 1. Executing the program on this database generates a series of `position(P,I)` assertions with  $P$  representing the position of integer  $I$ .

The time required to sort an unsorted database containing the numbers 1 through  $n$  is given in the table below. The column captioned with “Fib” is the normal version, where Fibonacci heaps are used as a priority queue. The algorithm sorts the numbers in  $\mathcal{O}(n \log n)$  time. The column captioned with “Static” is a version where the static priority queue is used instead. Since now the find-min and extract-min operations take time proportional to the number of different priorities, this leads to a quadratic behavior. Finally, the column captioned with “Static mm” uses a version of the static priority queue where

<sup>7</sup> The worst case depends on the order in which rule instances with equal priority are fired.

the minimum is maintained. This reduces the cost of the find-min operation to constant time. Moreover, since the elements are extracted in order from 1 to  $n$ , each extract-min operation takes constant time as well for this particular case (and for any case where the range of numbers to sort is proportional to the amount of numbers to sort). This in fact results in our sorting algorithm to behave linearly.

$n$	Static mm	Fib	Static
32	0.04	0.05	0.07
64	0.07	0.11	0.13
128	0.15	0.22	0.43
256	0.32	0.47	1.48
512	0.62	0.98	5.38
1024	1.25	2.04	20.59
2048	2.48	4.33	79.81
4096	4.98	9.10	310.77
8192	9.97	18.99	1236.92
16384	19.96	40.12	4997.63

## D Fibonacci Heaps with One Node per Priority

Fibonacci heaps can be implemented such that items with equal priority share a node in every heap. Each node consists of a double linked list of equal priority items. A hash table can be used to find a node given its heap and priority. We assume that all hash table operations (insertion, deletion and lookup) take constant time. Fibonacci heaps with one node per priority still support insertion and finding a minimal priority item in constant time (amortized). The cost of deleting an item decreases from  $\mathcal{O}(\log n)$  with  $n$  the number of items in the heap, to  $\mathcal{O}(\log N)$  with  $N$  the number of distinct priorities (i.e. the number of nodes in the heap).

Merging two heaps becomes more complicated. After such an operation, a given priority can be represented by two nodes which need to be merged. The obvious strategy is to traverse one of the heaps (the one with the least number of items) and inserting its items into the other heap. If  $N$  is the number of nodes in the smallest heap, then the cost of merging is  $\mathcal{O}(N)$ , i.e. it only depends on the number of nodes and not on the number of items. The following theorem gives a bound on the total cost of heap merging for a set of heaps, given a limit on the accumulated number of items over all heaps in the set, or in other words, the number of inserts.

**Theorem 5.** *For a given set of Fibonacci heaps with one node per priority in which  $n$  items are inserted with  $N$  distinct priorities in total, the total cost of merging heaps from this set is  $\mathcal{O}(n \log N)$ .*

*Proof.* The total cost is proportional to the number of nodes copied. If there are  $n$  items in total, spread over all heaps, and since every node contains at least

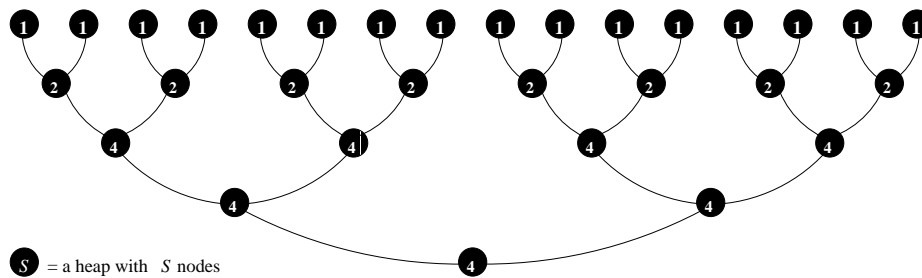
one item, there can be at most  $n$  nodes initially. Because we copy the smallest heap into the largest one, every time a node is copied, the size of its heap at least doubles. This means that the heap of a node that has been copied  $k$  times contains at least  $2^k$  items, and so there can be at most  $\max(N \cdot n/2^k, n)$  such nodes.<sup>8</sup> For  $k \leq \log N$ , there are at most  $n$  nodes that have been copied  $k$  times. For  $k > \log N$ , there are at most  $N \cdot n/2^k$  such nodes. No node can be copied more than  $\log n$  times because then it would be part of a heap with more than  $n$  items. In total, at most

$$\underbrace{n + n + \dots + n}_{\log N \text{ times}} + \frac{n}{2} + \frac{n}{4} + \dots + N$$

nodes are copied, and so the total cost for merging is  $\mathcal{O}(n \log N)$ .  $\square$

A worst case scenario in which this upperbound is reached, exists. This scenario consists of two phases. In the first phase, each heap contains as many items as nodes (i.e. every item has a unique priority). We start by pairwise merging  $n$  heaps with 1 node each, then  $n/2$  heaps with 2 nodes each, and so on until finally we have  $n/N$  heaps with each of them containing  $N$  nodes. The total cost of the first phase is  $\mathcal{O}(n \log N)$ . In the second phase, the remaining  $n/N$  heaps are merged. All intermediate heaps, as well as the final heap, contain  $N$  nodes, and so the total cost of the second phase is  $\mathcal{O}(n)$ . In conclusion, the total cost of merging a set of heaps with  $n$  items and  $N$  distinct priorities is  $\mathcal{O}(n \log N)$ .

*Example 5.* Let there be  $n = 16$  items with in total  $N = 4$  distinct priorities. The worst case merging scenario is shown in Figure 2. In the first phase, 16 heaps with 1 node each and 8 heaps with 2 nodes each are pairwise merged. There remain 4 heaps with each of them containing 4 nodes. In the second phase, a total of  $4 + 2 = 6$  heaps with 4 nodes each are pairwise merged until finally 1 heap with 4 nodes remains.



**Figure2.** Heap Merging: Worst Case Scenario

<sup>8</sup> Every heap contains at most  $N$  nodes and there can be at most  $n/2^k$  heaps that contain a node that has been copied  $k$  times.

## E Complexity of the Merge Sort Algorithm

In Section 4, a CHR<sup>IP</sup> implementation of the merge sort algorithm is translated into the following Logical Algorithms program:

```

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).

```

In this section, we show that the total runtime of the algorithm is  $\mathcal{O}(n \log n)$  if there initially are  $n$  `number/2` assertions.

No new `number/2` assertions are ever asserted. Rule `ms3` converts one `number/2` assertion into one `merge/3` assertion each time it fires. The number of (strong) prefix firings for rule `ms3` hence is  $\mathcal{O}(n)$ . Rule `ms2` decreases the number of `merge/3` assertions by one and so it can fire  $n - 1$  times. In any state, there are at most two positive `merge/3` assertions with the same first argument. This invariant holds in the initial state because there are no `merge/3` assertions in the input database and rule `ms2` can fire after each new `merge/3` assertion, enforcing the invariant. It limits the number of prefix firings for rule `ms2` to  $\mathcal{O}(n)$ .

Using similar reasoning it holds that in any state, there are at most two positive `arrow/3` assertions with the same first argument. Two numbers  $X_1$  and  $X_m$  are connected by a *chain* of length  $m - 1$  if the following assertions hold: `arrow(X1,X2,_)`, `arrow(X2,X3,_)`, ..., `arrow(Xm-1,Xm,_)`. At priority 2 it holds that for every `merge(N,X,_)` assertion, the maximal length of a chain starting in  $X$  is  $N$ . Indeed, this holds for `merge(0,_,_)` assertions and if it holds for `merge(N,_,_)` assertions, it also holds for `merge(2 · N + 1,_,_)` assertions, because two chains of length  $N$  are linked with an extra `arrow/3` assertion and merged by up to  $2 · N$  firings of rule `ms1`.

Two `merge(N,_,_)` assertions are combined into a `merge(2 · N + 1,_,_)` assertion, so the  $n$  `merge(0,_,_)` assertions generated by rule `ms3` generate  $n/2$  `merge(1,_,_)` assertions, which in turn are combined into  $n/4$  `merge(3,_,_)` assertions and so on until finally 1 `merge(n - 1,_,_)` assertion is generated. The sum of all  $N$  in these `merge(N,_,_)` assertions is  $\mathcal{O}(n \log n)$ . Rule `ms1` fires  $\mathcal{O}(N)$  times after every new `merge(N,_,_)` assertion and because there are at most two positive `arrow/3` assertions with the same first argument, there are  $\mathcal{O}(n \log n)$  (strong) prefix firings of rule `ms1`.

In conclusion, for an input database consisting of  $n$  `number/2` assertions, there are  $\mathcal{O}(n \log n)$  prefix firings for rule `ms1`,  $\mathcal{O}(n)$  prefix firings for rule `ms2` and  $\mathcal{O}(n)$  prefix firings for rule `ms3`. Using the meta-complexity theorem, the total runtime is  $\mathcal{O}(n \log n)$ .

**The “As Time Goes By” Approach** In [5,7], an upperbound on the worst case time complexity of a program  $P$  is given as

$$\mathcal{O} \left( D \sum_{r \in P} (c_{max}^{n_r} (O_{H_r} + O_{G_r}) + (O_{C_r} + O_{B_r})) \right)$$

where  $D$  is the maximal derivation length (i.e. the maximal number of rule firings),  $c_{max}$  is the maximal number of CHR constraints in the store, and for each rule  $r \in P$ :

- $n_r$  is the number of heads in  $r$
- $O_{H_r}$  is the cost of head matching, i.e. checking that a given sequence of  $n_r$  constraints match with the  $n_r$  heads of rule  $r$
- $O_{G_r}$  is the cost of checking the guard
- $O_{C_r}$  is the cost of adding built-in constraints after firing
- $O_{B_r}$  is the cost of adding and removing CHR constraints after firing

Using a similar analysis as for the Logical Algorithms translation, we can derive that  $D = \mathcal{O}(n \log n)$  and  $c_{max} = \mathcal{O}(n)$  where  $n$  is the number of `number/1` constraints in the query.<sup>9</sup> The cost of head matching ( $O_{H_r}$ ), guard checking ( $O_{G_r}$ ), adding built-in constraints ( $O_{C_r}$ ), and adding and removing CHR constraints ( $O_{B_r}$ ), can all be assumed constant. The number of heads  $n_r$  of a rule  $r \in P$  is at most 2. Filling in these numbers, we derive a total worst case complexity of  $\mathcal{O}(n^3 \log n)$ .

---

<sup>9</sup> In Theorem 4.2 of [7] a worst case upperbound of  $c_{max} = \mathcal{O}(c + D)$  is used, with  $c$  the number of constraints in the query, which becomes  $c_{max} = \mathcal{O}(n \log n)$  in this example. The bound we use is tight, i.e.  $c_{max} = \Theta(n)$ .