Implementation of Logical Retraction in Constraint Handling Rules with Justifications

Thom Frühwirth

Ulm University, Germany
thom.fruehwirth@uni-ulm.de

Abstract. In previous work we added justifications to Constraint Handling Rules (CHR) to enable logical retraction of constraints for dynamic algorithms. We presented a straightforward source-to-source transformation to implement this conservative extension. In this companion paper, we improve the performance of the transformation. We discuss its worst-case time complexity in general. Then we perform experiments. We benchmark the dynamic problem of maintaining shortest paths under addition and retraction of paths. The results validate our complexity considerations.

1 Introduction

Justifications have their origin in truth maintenance systems (TMS) [McA90] for automated reasoning. Derived information (a formula) is explicitly stored and associated with the information it originates from by means of justifications. With the help of justifications, conclusions can be withdrawn (undone) by retracting their premises. By this *logical retraction*, inconsistencies can be repaired by retracting one of the reasons for the inconsistency.

In the formalism and programming language Constraint Handling Rules (CHR) [Frü09,Frü15], conjunctions of atomic formulae (constraints) are rewritten by rule applications. When algorithms are written in CHR, constraints represent both data and operations. CHR is already incremental by nature, i.e. constraints can be added at runtime. Logical retraction then adds decrementality. To accomplish logical retraction in CHR, we have to be aware that CHR constraints can also be deleted by rule applications. These constraints may have to be restored when a premise constraint is retracted. With logical retraction, any algorithm written in CHR becomes *fully dynamic*[1]. Operations can be undone and data can be removed at any point in the computation without compromising the correctness of the result.

In [Fru17], we formally defined a correct conservative extension of CHR with justifications (CHR$^{\mathcal{J}}$). We gave a straightforward source-to-source transformation that adds justifications for user-defined constraints. A scheme of two rules sufficed to allow for logical retraction (deletion, removal) of constraints during

[1] Dynamic algorithms for dynamic problems should not be confused with dynamic programming.

computation. Without the need to recompute from scratch, these rules retract not only the constraint but also undo all consequences of the rule applications that involved the constraint.

The runtime performance of the previous translation scheme is not optimal, however. In this paper, we present an improved source-to-source transformation for logical retraction of constraints with justifications in CHR (CHR$^{\mathcal{J}}$). This transformation only imposes a constant factor overhead as long as justifications are not used for retraction. We will argue that the worst-case time complexity for any number of retractions is in general proportional to the number of rule applications, i.e. derivation length. The complexity of an algorithm expressed in CHR is usually a polynomial in the derivation length. Therefore retraction indeed has typically less complexity than recomputation from scratch at the expense of storing removed constraints. The added space complexity is again bounded by the derivation length. In our experiments, we will consider the dynamic problem of maintaining shortest paths under addition and retraction of paths.

Minimum Example. Given a multiset of numbers represented as conjunction `min(`n_1`),min(`n_2`),...,min(`n_k`)`. The constraint (predicate) `min(`n_i`)` means that the number n_i is a candidate for the minimum value. The following CHR rule filters the candidates.

```
min(N) \ min(M) <=> N=<M | true.
```

The rule consists of a left-hand side, on which a pair of constraints has to be matched, a guard check `N=<M` that has to be satisfied, and an empty right-hand side denoted by `true`. In effect, the rule takes two `min` candidates and removes the one with the larger value (constraints after the `\` symbol are deleted). Note that the `min` constraints behave both as operations (removing other constraints) and as data (being removed).

CHR rules are applied exhaustively. Here the rule keeps on going until only one, thus the smallest value, remains as single `min` constraint, denoting the current minimum. If another `min` constraint is added during the computation, it will eventually react with a previous *min* constraint, and the correct current minimum will be computed in the end. Thus the algorithm as implemented in CHR is incremental. It is not decremental, though: We cannot logically retract a *min* candidate. While removing a candidate that is larger than the minimum would be trivial, the retraction of the minimum itself requires to remember all deleted candidates and to find their minimum. As we will see, with the help of justifications, this logical retraction will be possible automatically.

Related Work. The work of Armin Wolf on Adaptive CHR [WGG00] introduced justifications into CHR. Different to our work, this technically involved approach requires to store detailed information about the rule instances that have been applied in a derivation in order to undo them. Adaptive CHR had a low-level implementation in Java [Wol01], while we give an implementation in CHR itself by source-to-source transformations.

The more recent work of Gregory Duck [Duc12] introduces SMCHR, a tight integration of CHR with a Boolean Satisfiability (SAT) solver for quantifier-free formulae including disjunction and negation as logical connectives. It is mentioned that for clause generation, SMCHR supports justifications for constraints.

Overview of the Paper. In the next section we recall abstract syntax and refined operational semantics for CHR. In Section 3, we describe CHR with justifications for logical retraction of constraints and its previous implementation by a straightforward source-to-source transformation. In Section 4, our current work is to optimize this implementation and to discuss its worst-case run-time complexity. In Section 5, we report on the results of experiments with our new implementation for the dynamic problem of maintaining shortest paths in a graph under addition (insertion) and deletion (retraction) of paths. The paper ends with conclusions and directions for future work.

2 Preliminaries

We recall abstract syntax and refined operational semantics of CHR [Frü09] in this section.

2.1 Abstract Syntax of CHR

Constraints are relations, distinguished predicates of first-order predicate logic. We differentiate between two kinds of constraints: *built-in (pre-defined) constraints* and *user-defined (CHR) constraints* which are defined by the rules in a CHR program.

Definition 1. A *CHR program* is a finite set of rules. A *(generalized) simpagation rule* is of the form

$$r : H_1 \backslash H_2 \Leftrightarrow C|B$$

where r : is an optional *name* (a unique identifier) of a rule. In the rule *head* (left-hand side), H_1 and H_2 are conjunctions of user-defined constraints, the optional *guard* C is a conjunction of built-in constraints, and the *body* (right-hand side) B is a goal. A *goal* is a conjunction of built-in and user-defined constraints. A *state* is a goal. Conjunctions are understood as *multisets* of their conjuncts.

In the rule, H_1 are called the *kept constraints*, while H_2 are called the *removed constraints*. At least one of H_1 and H_2 must be non-empty. If H_1 is empty, the rule corresponds to a *simplification rule*, also written

$$s : H_2 \Leftrightarrow C|B.$$

If H_2 is empty, the rule corresponds to a *propagation rule*, also written

$$p : H_1 \Rightarrow C|B.$$

In this work, we restrict given CHR programs to rules without built-in constraints in the body except *true* and *false*.

2.2 Operational Semantics of CHR

We follow the exposition in [SF06] in this subsection. Given a query, the rules of the program are applied to exhaustion. A rule is applicable, if its head constraints are matched by constraints in the current goal one-by-one and if, under this matching, the guard check of the rule holds. More formally, the guard is logically implied by the built-in constraints in the goal. Any of the applicable rules can be applied, and the application cannot be undone, it is committed-choice (in contrast to Prolog). When a simplification rule is applied, the matched constraints in the current goal are replaced by the body of the rule, when a propagation rule is applied, the body of the rule is added to the goal without removing any constraints. When a simpagation rule is applied, only the head constraints right to the backslash symbol are removed, the head constraints before are kept.

As in Prolog, almost all CHR implementations execute queries from left to right and apply rules top-down in the textual order of the program. This behavior has been formalized in the so-called refined semantics that was also proven to be a concretization of the standard operational semantics [DSdlBH04]. In this refined semantics of actual implementations, a CHR constraint in a query can be understood as a procedure that goes efficiently through the rules of the program in the order they are written, and when it matches a head constraint of a rule, it will look for the other, *partner constraints* of the head in the *constraint store* and check the guard until an applicable rule is found. We consider such a constraint to be *active*. If the active constraint has not been removed after trying all rules, it will be delayed and put into the constraint store as data. Constraints from the store will be reconsidered (woken) if newly added built-in constraints constrain variables of the constraint, because then rules may become applicable since their guards are now implied.

Hash Indexing in CHR. To achieve optimal time complexity, (near) constant-time addition, finding and removal of CHR constraints is required. Most current CHR libraries in Prolog are based on the KU Leuven CHR system. It supports indexes for terms via attributed variables, and in SWI Prolog also hash tables for ground terms and arrays for dense integers. The HAL CHR system and few other implementations also feature balanced trees, which are usually somewhat slower than hash tables. The hash table based indexes in SWI Prolog work at the argument level. In other words, for efficient constraint lookups, these arguments have to be ground during computation. Thus, for optimal performance, the SWI Prolog CHR system depends on mode and type information specified in constraint declarations.

3 CHR with Justifications (CHR$^{\mathcal{J}}$)

We present a conservative extension of CHR by justifications following [Fru17]. If justifications are not used, programs behave as without them. Justifications annotate atomic CHR constraints. A straightforward source-to-source transformation extends the rules with justifications.

3.1 CHR with Justifications for Logical Retraction

We start with adding justifications to CHR constraints and states.

Definition 2 (CHR Constraints and Initial States with Justifications). A *justification* f is a unique identifier. Given an atomic CHR constraint G, a *CHR constraint with justifications* is of the form G^F, where F is a set of justifications. An *initial state with justifications* is of the form $\bigwedge_{i=1}^{n} G_i^{\{f_i\}}$ where the f_i are distinct justifications.

We now define a source-to-source translation from rules to rules with justifications. Let *kill* (retract) and *rem* (remember removed) be to unary *reserved* CHR constraint symbols. This means they are only allowed to occur in rules as specified in the following.

Definition 3 (Translation to Rules with Justifications). Given a generalized simpagation rule

$$r : \bigwedge_{i=1}^{l} K_i \setminus \bigwedge_{j=1}^{m} R_j \Leftrightarrow C \mid \bigwedge_{k=1}^{n} B_k$$

Its translation to a *simpagation rule with justifications* is of the form

$$rf : \bigwedge_{i=1}^{l} K_i^{F_i} \setminus \bigwedge_{j=1}^{m} R_j^{F_j} \Leftrightarrow C \mid \bigwedge_{j=1}^{m} rem(R_j^{F_j})^F \wedge \bigwedge_{k=1}^{n} B_k^F \text{ where } F = \bigcup_{i=1}^{l} F_i \cup \bigcup_{j=1}^{m} F_j.$$

The translation ensures that the head and the body of a rule mention exactly the same justifications. The reserved CHR constraint *rem/1* (remember removed) stores the constraints removed by the rule together with their justifications.

Shorthand Notation. By abuse of notation, let $A^{\mathcal{J}}, B^{\mathcal{J}}, C^{\mathcal{J}} \ldots$ be conjunctions or corresponding states whose atomic CHR constraints are annotated with justifications according to the above definition of the rule scheme. Similarly, let $rem(R)^{\mathcal{J}}$ denote the conjunction $\bigwedge_{j=1}^{m} rem(R_j^{F_j})^F$.

We showed previously that rule applications correspond to each other in standard CHR and in CHR$^{\mathcal{J}}$.

Lemma 1 (Equivalence of Program Rules). *[Fru17]* There is a computation step $S \mapsto_r T$ with simpagation rule

$$r : H_1 \backslash H_2 \Leftrightarrow C | B$$

if and only if there is a computation step with justifications $S^{\mathcal{J}} \mapsto_{rf} T^{\mathcal{J}} \wedge rem(H_2)^{\mathcal{J}}$ with the corresponding simpagation rule with justifications

$$rf : H_1^{\mathcal{J}} \backslash H_2^{\mathcal{J}} \Leftrightarrow C | rem(H_2)^{\mathcal{J}} \wedge B^{\mathcal{J}}.$$

Since computations are sequences of connected computation steps, this lemma implies that computations in standard CHR program and in CHR$^{\mathcal{J}}$ correspond to each other. Thus CHR with justifications is a conservative extension of CHR.

Logical Retraction Using Justifications. We use justifications to retract a CHR constraint from a computation without the need to recompute from scratch. This means that all its consequences due to rule applications it was involved in are undone. CHR constraints added by those rules are removed and CHR constraints removed by the rules are re-added (inserted). To specify and implement this behavior, we give a scheme of two rules, one for retraction and one for re-adding of constraints. The reserved CHR constraint $kill(f)$ (retract) undoes all consequences of the constraint with justification f.

Definition 4 (Rules for CHR Logical Retraction). For each n-ary CHR constraint symbol c (except the reserved *kill* and *rem*), we add a rule to kill constraints and a rule to revive removed constraints of the form:

$$\text{kill} : kill(f) \setminus G^F \Leftrightarrow f \in F \mid true$$

$$\text{revive} : kill(f) \setminus rem(G^{F_c})^F \Leftrightarrow f \in F \mid G^{F_c},$$

where $G = c(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are different variables.

Note that a constraint may be revived and subsequently killed. This is the case when both F_c and F contain the justification f.

We proved previously correctness of logical retraction: the result of a computation with retraction is the same as if the constraint would never have been introduced in the computation. We showed that given a computation starting from an initial state with a *kill(f)* constraint that ends in a state where the *kill* and *revive* rules have been applied to exhaustion, then there is a corresponding computation without constraints that contain the justification f.

Theorem 1 (Correctness of Logical Retraction). *[Fru17]* Given a computation

$$A^{\mathcal{J}} \wedge G^{\{f\}} \wedge kill(f) \mapsto^* B^{\mathcal{J}} \wedge rem(R)^{\mathcal{J}} \wedge kill(f) \not\mapsto_{kill,revive},$$

where f does not occur in $A^{\mathcal{J}}$. Then there is a computation without $G^{\{f\}}$ and $kill(f)$

$$A^{\mathcal{J}} \mapsto^* B^{\mathcal{J}} \wedge rem(R)^{\mathcal{J}}.$$

3.2 Previous Implementation

We recall the implementation of [Fru17] for CHR with justifications (CHR$^{\mathcal{J}}$).

Constraints with Justifications. CHR constraints annotated by a set of justifications are realized by a binary infix operator `##`, where the second argument is a list of justifications:

$C^{\{F_1,F_2,\ldots\}}$ is realized as `C ## [F1,F2,...]`.

For convenience, we add rules that add a new justification to a given constraint `C`. For each constraint symbol `c` with arity `n` there is a rule of the form

```
addjust @ c(X1,X2,...Xn) <=> c(X1,X2,...Xn) ## [_F].
```

where the arguments of `X1,X2,...Xn` are different variables.

Rules with Justifications. A CHR simpagation rule with justifications is realized as follows:

$$rf : \bigwedge_{i=1}^{l} K_i^{F_i} \setminus \bigwedge_{j=1}^{m} R_j^{F_j} \Leftrightarrow C \mid \bigwedge_{j=1}^{m} rem(R_j^{F_j})^F \wedge \bigwedge_{k=1}^{n} B_k^F \text{ where } F = \bigcup_{i=1}^{l} F_i \cup \bigcup_{j=1}^{m} F_j$$

```
rf @ K1 ## FK1,... \ R1 ## FR1,... <=> C |
    union([FK1,...FR1,...],Fs), rem(R1##FR1) ## Fs,...B1 ## Fs,...
```

where the auxiliary predicate `union/2` computes the ordered duplicate-free union of a list of lists[2].

Rules kill, remove *and* revive. Justifications are realized as *flags* that are initially unbound logical variables. This eases the generation of new unique justifications and their use in killing. Concretely, the reserved constraint $kill(f)$ is realized as built-in equality `F=r`, i.e. the justification variable gets bound. If $kill(f)$ occurred in the head of a *kill* or *revive* rule, it is moved to the guard as equality test `F==r`.

revive : $kill(f) \setminus rem(C^{F_c})^F \Leftrightarrow f \in F \mid C^{F_c}$
kill : $kill(f) \setminus C^F \Leftrightarrow f \in F \mid true$

```
revive @ rem(C##FC) ## Fs <=> member(F,Fs),F==r | C ## FC.
remove @ C ## Fs <=> notfunctor(C,rem),member(F,Fs),F==r | true.
```

The check `notfunctor(C,rem)` ensures that `C` is not a `rem` constraint. The check for set membership in the guards is expressed using the standard nondeterministic Prolog built-in predicate `member/2`.

Logical Retraction with `killc/1`. We extend the translation to allow for retraction of derived constraints. The constraint `killc(C)` logically retracts constraint `C`. The two rules `killc` and `killr` try to find the constraint `C` - also when it has been removed and is now present in a `rem` constraint. The associated justifications point to all initial constraints that where involved in producing the constraint `C`. For retracting the constraint, it is sufficient to remove one of its producers. This introduces a choice implemented by the `member` predicate.

```
killr @  killc(C), rem(C ## FC) ## _Fs <=> member(F,FC),F=r.
killc @  killc(C), C ## Fs <=> member(F,Fs),F=r.
```

Note that in the first rule, we bind a justification `F` from `FC`, because `FC` contains the justifications of the producers of constraint `C`, while `Fs` also contains those that removed it by a rule application.

[2] More precisely, a simplification rule is generated if there are no kept constraints and a propagation rule is generated if there are no removed constraints.

Dynamic Minimum Example. Translating the minimum rule to one with justifications results in:

```
min(A)##B \ min(C)##D <=> A<C | union([B,D],E), rem(min(C)##D)##E.
```

The following shows an example query and the resulting answer in SWI-Prolog:

```
?- min(1)##[A], min(0)##[B], min(2)##[C].
rem(min(1)##[A])##[A,B], rem(min(2)##[C])##[B,C], min(0)##[B].
```

The constraint `min(0)` remained. This means that `0` is the minimum. The constraints `min(1)` and `min(2)` have been removed and are now remembered. Both have been removed by the constraint with justification `B`, i.e. `min(0)`.

We now logically retract with `killc` the constraint `min(1)` at the end of the query. The `killr` rule applies and removes `rem(min(1)##[A])##[A,B]`. (In the rule body, the justification `A` is bound to `r` - to no effect, since there are no other constraints with this justification.)

```
?- min(1)##[A], min(0)##[B], min(2)##[C], killc(min(1)).
rem(min(2)##[C])##[B,C], min(0)##[B].
```

What happens if we retract the current minimum `min(0)`? The constraint `min(0)` is removed by binding justification `B`. The two `rem` constraints for `min(1)` and `min(2)` involve `B` as well, so these two constraints are re-introduced and react with each other. Note that `min(2)` is now removed by `min(1)` (before it was `min(0)`). The result is the updated minimum, which is `1`.

```
?- min(1)##[A], min(0)##[B], min(2)##[C], killc(min(0)).
rem(min(2)##[C])##[A,C], min(1)##[B].
```

4 Optimizing the Implementation

We would like to avoid any overhead complexity-wise when computing with justifications as long as we do not use them for retraction. We are ready to accept a constant factor penalty. While the insertion of `rem` constraints takes constant time, the computation of the union of justifications is linear in the sizes of its input justification sets. The idea is to delay this computation until it is needed due to a retraction. We actually never compute the union of justifications, but will use the `union` constraints as data to find the necessary justifications. We describe the modifications for this new implementations and then discuss the complexity of this approach.

4.1 New Improved Implementation

To retract a constraint with justification `F`, the constraint `killd(F)` (kill down) finds its initial justifications. The arguments of the delayed `union` constraints are unbound variables now (except for the singleton sets of the justifications from the initial constraints in the query). The constraint `killd` has to find the `union`

constraint with its justification in the output and follow all its input justifications (which are represented by a list). It proceeds recursively with the help of `killl` (kill list) until it reaches an initial justification. On the way, we can stop if we see a justification again that we have already seen.

```
already_seen  @ killd(F) \ killd(F) <=> true.
go_to_initial @ union(FL,F) \ killd(F) <=> killl(FL).

        killl @  killl([]) <=> true.
        killl @  killl([F|FL]) <=> killd(F), killl(FL).
```

Then the auxiliary constraint `killone` (kill one) chooses one of these justifications in turn and removes it.

```
choice @ killone, killd([F]) <=> (F=r,waitrem ; killone).
done @   killone <=> false.
```

The rule `choice` uses Prolog's disjunction in the body. In the first disjunct, the binding of justification `F` to the constant `r` marks it as to be killed and wakes up all constraints in which this justification occurs. In this way, constraints are retracted and revived, respectively. The auxiliary constraint `waitrem` delays re-addition of previously removed constraints via the rule `revive` until all constraints have been retracted by the `remove` rule. This improves the performance. The recursion on `killone` in the second disjunct ensures that all justifications are eventually tried. Note that as a consequence, in rule `done` we must fail (not succeed), since we then have exhausted trying all justifications.

Now we also have to kill all output justifications of unions that have this killed justification as input justification, i.e. we go upwards.

```
go_upwards @  union(FL,F) <=> member(F1,FL),F1==[r] | F=[r].
```

Note that we will only pass a subset of the `union` constraints that `killd` visited, those that involve the chosen initial justification. We will also pass additional other `union` constraints as consequence of this.

Finally, for retraction, we remove constraints with killed justifications and we revive remembered constraints with killed justifications. We translate program constraints `C` with justifications `F` of the form `c(X1,..Xn)##F` into `c(X1,..Xn,F)` to support argument-wise indexing if necessary.

```
remove @ c(X1,..Xn,[r]) <=> true.
revive @ waitrem \ rem(c(X1,..Xn,FC),[r]) <=> c(X1,..Xn,FC).
         waitrem <=> true.
```

Here we put `waitrem` to work to trigger the re-addition of constraints in the `revive` rule. Having done so, `waitrem` is removed at the very end.

4.2 Worst-Case Time Complexity

We now discuss the complexity of our optimized implementation in terms of the input size and derivation length following the principles of [Frü02]. Let k be the largest number of head constraints in a given program. Note that k is a constant. Let c be the number of CHR constraints in the initial state (query). Let n be the derivation length of a computation, i.e. the number of rule applications (transitions).

The complexity of the original computation is at least n, because there are n rule applications that take at least constant time each. If the computation does not fail, each initial constraint is processed, which adds c to the lower bound of the complexity, which thus is $n + c$. Typically, n is larger than c, so we may assume just n.

All rule tries (application attempts) and rule applications take constant time, mostly because of the index on the justification. There is no overhead in runtime complexity until a constraint is killed: the `union` constraint and the `rem` constraints are just added to the constraint store. Since the number of `rem` constraints is bounded by k, complexity does not increase, if constraints can be added (inserted) in constant time. Based on these observations, we can also see that the space complexity is bounded by $O(n)$.

The `union` constraints have at most k input justifications that already have been introduced. The result is the output justification, represented by a new fresh logical variable. The `union` constraints form a directed acyclic graph (dag) with bounded width k, where the nodes are the justification set variables and where there is an directed edge (arc) from each input to the output justification for each union in a derivation. Since the output justification is always new, the corresponding graph is acyclic. It is typically not a tree, since a union may have input justifications from arbitrary previous unions.

There are at most n unions in a computation of length n. Thus there are at most n new justification nodes and c initial justifications. Therefore we have at most $n+c$ different nodes. The number of edges is at most k for each union and is therefore of order $O(n)$. The constraint `killd` has to go along at most kn edges, pass at most $n + c$ different nodes and stop at most at c initial justifications.

The constraint `killone` will chose the next initial justification in constant time. There may be up to c choices. Once we have chosen this initial justification to use for killing and retraction, we use the rule `go_upwards` to find all effected justifications with the help of the `union` constraint. We may have up to n non-initial justifications to revive and remove (kill) constraints in turn. Typically, the number will be much smaller, because n refers to all union constraints in the derivation. For each justification, there can only be a bounded number of remembered (k) and added constraints, because the number of head and body constraints in rules is bounded in a given program.

The killing of a justification and the retraction of constraints is accomplished by binding the justification variable. This will wake up all constraints in which the variable occurs. These are the `union` constraints and the all program constraints that have this justification. Thus the rule `go_upwards` and `remove` are

immediately applicable, while the `revive` rule applications have to wait for the constraint `waitrem`.

In summary, the overall worst-case complexity of retracting a constraint with one choice of an initial justification is of order $O(n)$ (assuming $n > c$). The complexity trying each of the up to c found initial constraints is then $O(nc)$. Note that the complexity of removing all constraints or all initial c justifications in a computation is also bounded by $O(n)$, since the number of remembered and added constraints is also of order $O(n)$.

The additional cost of processing the revived re-added constraints is of course dependent on the given program and has to be added to the above complexity results. In the worst case, it amounts to a complete recomputation from scratch (cf. minimum example). It may be constant in the best case. If all rules of the program can be tried and applied in constant time, the derivation length n that was needed for c initial constraints may provide a $O(n)$ worst case complexity for computations with the revived constraints, thus leaving the overall worst-case complexity at $O(n)$.

5 Experiments

Experiments were run with SWI Prolog 6.2.1. in standard configuration on an Apple Mac Mini with OS X 10.9.5 2,5 GHz Intel Core i5 and 4 GB RAM. For compilation of the CHR files debugging was switched off and full optimization enabled. We explicitly specified the arguments for indexing of program constraints in a declaration. This lead to a constant-factor improvement of the runtime over automatic indexing provided by the CHR compiler.

We also introduced `passive` declarations in the rules that handle the justifications for retraction where feasible. These annotate head constraints in rules. Such a constraint is then treated as data only that has to be searched for in the constraint store. No active code is generated for that constraint, i.e. it does not behave as an operation anymore that looks for its matching partner constraints. This optimizations avoids useless rule tries. Note that some of these passive constraints are also automatically inferred by the compiler.

The programs used can be found in the appendix of the full online version of this paper.

5.1 Dynamic All-Pair Shortest Paths

We want to find the shortest distance between all pairs of nodes in a complete directed graph whose edges are annotated with non-negative distances. Initially, for each edge, there is a corresponding path with the distance of the edge. For every other pair of nodes, the unknown distances are initialized with ∞. Then the following rule suffices to solve the problem:

```
shorten @ path(I,K,D1), path(K,J,D2) \ path(I,J,D3) <=>
                         D4 is D1+D2, D3>D4 | path(I,J,D4).
```

A currently shortest path between nodes `I` and `J` is replaced by the sum of the distances between paths `I` to `K` and `K` to `J` if this new distance is shorter. Note that the graph is complete. If the rule is not applicable anymore, all paths must be shortest. From the `shorten` rule we generated the following rules augmented with justifications

```
add_justification @ path(A,B,C) <=> path(A,B,C,[D]).

shorten @ path(A,B,C,D), path(B,E,F,G) \ path(A,E,H,I) <=>
       J is C+F,H>J |
       union([D,G,I],K), rem(path(A,E,H,I),K), path(A,E,J,K).
```

Example. The answer output has been slightly edited to improve readability.

```
?-path(a,b,1),path(b,a,2),path(a,c,3),path(c,a,0),path(b,c,1),path(c,b,4).
rem(path(c,b,4,[A]),B), rem(path(a,c,3,[C]),D), rem(path(b,a,2,[E]),F),
union([[G],[H],[A]],B), union([[H],[I],[C]],D), union([[I],[G],[E]],F),
path(c,b,1,B), path(a,c,2,D), path(b,a,1,F), path(b,c,1,[I]),
path(c,a,0,[G]), path(a,b,1,[H])
```

Initial justifications are in square brackets as single elements of lists. Thus the last three paths in the answer were not shortened, while the other three paths were shortened once, as can be seen by the deleted original `path/3` constraints for them. From the first arguments of the delaying `union/2` constraints we can also read off the constraints that lead to a shorter path.

For our experiments, the `shorten` rule was then instrumented to count rule tries (in the guard) and applications (in the body) with the help of Prolog's global variables. We explicitly added indexing information for the compiler because it slightly improved the performance on our examples. This means there is an hash index on the first and second argument of the `path/4` constraint and it can also be accessed without index.

Random Graph Generation and Shortest Paths. We generated complete graphs from a given number of nodes represented by integers. For every pair of different nodes, a path is generated with a random distance between 1 and the number of nodes. This is accomplished by the rule:

```
gengraph(N), node(A), node(B) ==> random(1,N,D), path(A,B,D).
```

In Figure 1 the number of nodes of the random directed graph is given, leading to a quadratic number of paths. Column *Apply* reports the number of applications of the `shorten` rule, while column *Try* shows how often this rule has been tried. Finally, *Time* reports the execution time in seconds. The time is roughly proportional to the number of rules tries indicating that indexing reduces the time for finding the three matching head constraints indeed to a constant.

Previous Implementation				New Implementation			
Nodes	Apply	Try	Time	Nodes	Apply	Try	Time
12	125	2817	0.208	12	157	2958	0.106
12	113	2332	0.171	12	129	2770	0.093
12	142	2567	0.206	12	99	2362	0.083
14	210	4929	0.494	14	246	5215	0.225
14	250	5564	0.590	14	248	4693	0.189
14	223	4274	0.467	14	270	5449	0.234
16	338	9105	1.218	16	366	7667	0.402
16	379	8607	1.299	16	333	7643	0.391
16	362	8425	1.234	16	356	7613	0.391
18	501	11256	2.154	18	499	11899	0.759
18	502	12799	2.390	18	476	10567	0.674
18	416	9915	1.693	18	404	9980	0.628
21	801	21171	5.965	21	837	19134	1.499
21	783	22265	5.970	21	858	23550	1.928
21	778	19831	5.502	21	830	21094	1.676
24	1318	38188	14.809	24	1228	36507	3.553
24	1295	40549	16.172	24	1165	32543	3.422
24	1162	31898	12.270	24	1316	42426	4.039

Fig. 1. Shortest Paths for Random Complete Directed Graphs

Complexity. Let v be the number of nodes in the graph. There can be at most v^2 shortest path, one between each pair of nodes, so $c = v^2$. With indexes on the nodes in a path, the rule `shorten` can be applied in constant time, given one of the path constraints. The worst-case derivation length depends on the scheduling of paths for rule application. The optimal complexity is $O(v^3)$ when the scheduling of the Floyd-Warshall algorithm is used. It assumes an order on nodes and processes paths by their smallest nodes. We do not specify the scheduling and therefore expect a higher polynomial complexity in v. To reach the optimal complexity was not the scope of this work, since here we are interested in increasing the performance of logical retraction in comparison with the previous implementation.

Back to our experiments reported in Figure 1: for a complete graph with v nodes and v^2 paths, the average execution time is of order $O(v^4)$ as was confirmed by computing the interpolating polynomial with WolframAlpha. This also holds for the number of rule tries and applications. So the derivation length n is quadratic in the number of paths c, i.e. $O(c^2)$. The previous implementation has a similar complexity, but a higher constant factor.

Logical Retraction of Paths. In Figure 2 we can see that the times for retracting all shortest paths in a complete random directed graph vary. The columns *Apply* and *Try* refer to accumulated recomputations of shortest paths after retraction of paths. *Down* reports the number of rule applications for going to the initial justifications through `union` constraints, while *Up* counts the propagation of the

New Implementation							
Nodes	Apply	Try	Down	Up	Remove	Revive	Time
12	30	481	369	167	196	167	0.032
12	37	581	208	147	180	147	0.030
12	17	541	78	97	119	97	0.026
14	42	832	1990	242	281	242	0.075
14	54	894	1592	278	317	278	0.076
14	77	1245	1939	318	350	318	0.095
16	111	1901	5490	491	539	491	0.203
16	55	1413	3481	383	428	383	0.141
16	135	2513	1496	447	508	447	0.193
18	132	3158	7735	595	663	595	0.341
18	96	1869	4810	514	581	514	0.215
18	199	4598	8323	657	732	657	0.465
21	180	4963	51274	962	1033	962	1.170
21	207	4671	203119	917	966	917	2.980
21	188	4559	75441	908	985	908	1.381

Fig. 2. Removing All Shortest Paths from Random Graphs

killed justification to the roots. The counts for *Down* and thus the time needed vary, the variation seems to increase the larger the graph is. This number depends on the number of updates to particular intermediate shortest paths, i.e on the depth of the justification dag.

Remove and *Revive* show the number of actual removals of constraints and re-addition of previously removed path constraints. These last two numbers are similar, with slightly more removals than revivals. (Note that re-added constraints may be removed afterwards.) The numbers for *Up* and *Revive* are identical, because the `shorten` rule always removes a single path constraint.

Overall, the complexity is once again quartic, $O(v^4)$. This corresponds once again to the derivation length and thus is in line with our complexity considerations in the previous section. It also means that the overhead of the recomputations is neglectable complexity-wise. Indeed, comparing the two figures, we can see that it typically takes less time to remove each shortest paths one by one and recompute all effected paths each time than to compute all the shortest paths initially. Moreover, the numbers of path recomputations are about a fourth of the number of initial path computations.

Note that recomputing from scratch would result in $O(v^2)$ recomputations (one for each retracted path) of complexity $O(v^4)$ each and thus in a polynomial of higher degree. The previous implementation also has a worse polynomial complexity for retracting constraints. For a graph of size 14, the previous implementation is already about an order of magnitude slower.

6 Conclusions

We presented an improved source-to-source transformation for logical retraction of constraints with justifications in CHR (CHR$^{\mathcal{J}}$). This transformation only imposes a constant factor overhead as long as justifications are not used for retraction. We argued that the worst-case time complexity for any number of retractions is in general proportional to the number of rule applications, i.e. derivation length. The complexity of an algorithm expressed in CHR is usually a polynomial in the derivation length. Therefore retraction indeed has typically less complexity than recomputation from scratch at the expense of storing removed constraints. The added space complexity is again bounded by the derivation length. In our experiments, we benchmarked the dynamic problem of maintaining shortest paths under addition and retraction of paths. The results verify our complexity considerations. For future work, we would like to further improve the implementation and benchmark it, taking care of proper indexing. At the same time, we would like to investigate how logical as well as classical algorithms like union-find behave when they become dynamic in CHR$^{\mathcal{J}}$.

References

[DSdlBH04] Gregory J. Duck, Peter J. Stuckey, Maria Garcia de la Banda, and Christian Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *20th International Conference on Logic Programming (ICLP)*, LNCS. Springer, 2004.

[Duc12] Gregory J Duck. Smchr: Satisfiability modulo constraint handling rules. *Theory and Practice of Logic Programming*, 12(4-5):601–618, 2012.

[Frü02] Thom Frühwirth. As time goes by II: More automatic complexity analysis of concurrent rule programs. In A. Di Pierro and H. Wiklicky, editors, *QAPL '01: Proc. First Intl. Workshop on Quantitative Aspects of Programming Languages*, volume 59(3):185–206 of *ENTCS*. Elsevier, 2002.

[Frü09] Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

[Frü15] Thom Frühwirth. Constraint handling rules – what else? In *Rule Technologies: Foundations, Tools, and Applications*, pages 13–34. Springer International Publishing, 2015.

[Fru17] Thom Fruehwirth. Justifications in Constraint Handling Rules for Logical Retraction in Dynamic Algorithms. *27th International Symposium on Logic-Based Program Synthesis and Transformation LOPSTR 2017*, 2017.

[McA90] David A McAllester. Truth maintenance. In *AAAI*, volume 90, pages 1109–1116, 1990.

[SF06] T. Schrijvers and T. Frühwirth. Optimal union-find in constraint handling rules, programming pearl. *Theory and Practice of Logic Programming (TPLP)*, 6(1), 2006.

[WGG00] Armin Wolf, Thomas Gruenhagen, and Ulrich Geske. On the incremental adaptation of chr derivations. *Applied Artificial Intelligence*, 14(4):389–416, 2000.

[Wol01] Armin Wolf. Adaptive constraint handling with chr in java. In *International Conference on Principles and Practice of Constraint Programming*, pages 256–270. Springer, 2001.