Preliminary version 2019.
Final version in Journal Fundamenta Informaticae, vol. 173 no. 4, IOS Press, 2020.

Justifications in Constraint Handling Rules for Logical Retraction in Dynamic Algorithms: Theory, Implementations, and Complexity

Thom Frühwirth

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

Abstract. We present a concise source-to-source transformation that introduces justifications for user-defined constraints into the rule-based Constraint Handling Rules (CHR) programming language. There is no need to introduce a new semantics for justifications. This leads to a conservative extension of the language, as we can show the equivalence of rule applications.
A scheme of two rules suffices to allow for logical retraction (deletion, removal) of CHR constraints during computation. Without the need to recompute from scratch, these rules remove the constraint and also undo all its consequences. The scheme applies to CHR rules without built-in constraints in their bodies. We prove a confluence result concerning the rule scheme. We prove its correctness in general and for confluent programs.
We give an implementation that is available online. We show its correctness for confluent programs. We present two classical examples of dynamic algorithms: maintaining the minimum of a changing set of numbers and shortest paths in a graph whose edges change.
Finally, we improve the efficiency of the implementation while maintaining correctness. The computational overhead of introducing justifications and of performing logical retraction, i.e. the additional time and space needed, is proportional to the derivation length in the original program. This overhead may increase space complexity, but does not change the worst-case time complexity.

Additional Keywords. Truth Maintenance, Rule-Based Programming, Logic Programming, Computational Logic, Source-to-Source Program Transformation, Constraint Deletion, Confluence.

1 Introduction

Justifications have their origin in truth maintenance systems (TMS) [McA90] for automated reasoning. In this knowledge representation method, derived information (a formula) is explicitly stored and associated with the information it originates from by means of justifications. This dependency can be used to explain the reason for a conclusion (consequence) by its initial premises. With the help of justifications, conclusions can be withdrawn by retracting their premises.

By this *logical retraction*, e.g. default reasoning can be supported and inconsistencies can be repaired by retracting one of the reasons for the inconsistency. An obvious application of justifications are dynamic constraint satisfaction problems (DCSP), in particular over-constrained ones [BM06].

In this work, we extend the applicability of logical retraction to arbitrary algorithms that are expressed in the programming language and formalism Constraint Handling Rules (CHR) [Frü09,Frü15,FR18]. In CHR, 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 of constraints, operations can be undone and data can be removed at any point in the computation without compromising the correctness of the result. Hence any algorithm written in CHR with justifications will become fully dynamic.

Minimum Example. Given a multiset of numbers `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. With the help of justifications, this logical retraction will be possible automatically.

Contributions and Overview. While there is previous work extending CHR with justifications (see section on related work), our approach of a source-to-source transformation leads to a straightforward implementation. A new semantics for justifications is not necessary. Our approach allows for strong proven correctness results and enables well-known CHR analysis techniques such as for confluence to be applied.

In the next section we recall abstract syntax and abstract as well as refined operational semantics for CHR. In this work, we restrict given CHR programs to

rules without built-in constraints in the body. This restriction is necessary as long as built-in constraint solvers do not support the removal of built-in constraints.

Then our contributions are as follows:

- We introduce CHR with justifications ($\text{CHR}^{\mathcal{J}}$) in Section 3. We enhance standard CHR programs with justifications by a source-to-source program transformation. We prove the operational equivalence of rule applications in both settings. Thus $\text{CHR}^{\mathcal{J}}$ is a conservative extension of standard CHR.
- We define a scheme of two rules to enable logical retraction of constraints based on justifications in Section 4. We show that the rule scheme is confluent with each rule in any given program, independent of the confluence of that program. We show that logical retraction can undo rule applications. We prove correctness of logical retraction in general and for confluent programs: the result of a computation with retraction is the same as if the retracted constraint was never introduced.
- We present an implementation of $\text{CHR}^{\mathcal{J}}$ in CHR and Prolog (available online) in Section 5 and show its correctness for confluent programs. We discuss two classical examples for dynamic algorithms, maintaining the minimum of a changing set of numbers and maintaining shortest paths in a graph whose edges change.
- In Section 6 we optimize this implementation and show that correctness still holds. In Section 7 we show its worst-case space and time complexity. We prove that the costs of introduction of justifications and of logical retraction are bounded by the derivation length in the original program.

The paper ends with discussion of related work in Section 8 and with conclusions and directions for future work.

This paper is based on the approach of [Fru17b] and the improved implementation of [Fru17a]. In particular the correctness and complexity theorems appear here for the first time.

2 Preliminaries

We recall the abstract syntax and the equivalence-based abstract operational semantics of CHR in this section. We also informally describe the refined operational semantics typically realized in sequential implementations of CHR. For parallel and concurrent CHR see [Fru18].

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. The built-in solver stores, combines, and simplifies the built-in constraints. Built-in constraints can be used as tests in the guard as well as for auxiliary computations in the body of a rule.

Upper-case letters stand for (possibly empty) conjunctions of constraints in this paper.

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

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

2.2 Abstract Operational Semantics of CHR

Computations in CHR are sequences of rule applications. The operational semantics of CHR is given by the state transition system. It relies on a structural equivalence between states that abstracts away from technical details in a transition [RBF09,Bet14].

State equivalence treats built-in constraints semantically and user-defined constraints syntactically. Basically, two states are equivalent if their built-in constraints are logically equivalent (imply each other) and their user-defined constraints form syntactically equivalent multisets. For example,

$$X{=}{<}Y \wedge Y{=}{<}X \wedge c(X,Y) \;\equiv\; X{=}Y \wedge c(X,X) \not\equiv X{=}Y \wedge c(X,X) \wedge c(X,X).$$

For a state S, the notation S_{bi} denotes the built-in constraints of S and S_{ud} denotes the user-defined constraints of S.

Definition 2 (State Equivalence). Two states $S_1 = (S_{1bi} \wedge S_{1ud})$ and $S_2 = (S_{2bi} \wedge S_{2ud})$ are *equivalent*, written $S_1 \equiv S_2$, if and only if

$$\models \forall(S_{1bi} \rightarrow \exists\bar{y}((S_{1ud} = S_{2ud}) \wedge S_{2bi})) \wedge \forall(S_{2bi} \rightarrow \exists\bar{x}((S_{1ud} = S_{2ud}) \wedge S_{1bi}))$$

with $\bar{x}$ those variables that only occur in S_1 and $\bar{y}$ those variables that only occur in S_2.

Using this state equivalence, the abstract CHR semantics is defined by a single transition (computation step). It defines the application of a rule. Note that CHR is a committed-choice language, i.e. there is no backtracking in the rule applications.

Definition 3 (Transition). Let the rule $(r : H_1 \backslash H_2 \Leftrightarrow C|B)$ be a variant[1] of a rule from a given program $\mathcal{P}$. The *transition (computation step)* $S \mapsto_r T$ is defined as follows, where S is called *source state* and T is called *target state*:

$$\frac{S \equiv (H_1 \wedge H_2 \wedge C \wedge G) \quad (r : H_1 \backslash H_2 \Leftrightarrow C|B) \in \mathcal{P} \quad (H_1 \wedge C \wedge B \wedge G) \equiv T}{S \mapsto_r T}$$

The goal G is called *context* of the rule application. It remains unchanged. It is arbitrary and may be empty.

A *computation (derivation)* of a goal S in a program $\mathcal{P}$ is a connected sequence $S_i \mapsto_{r_i} S_{i+1}$ beginning with the *initial state (query)* S_0 that is S and ending in a *final state (answer, result)* or the sequence is *non-terminating (diverging)*. We may drop the reference to the rules r_i to simplify the presentation. The relation $\mapsto^*$ denotes the reflexive and transitive closure of $\mapsto$.

If the source state can be made equivalent to a state that contains the head constraints and the guard built-in constraints of a variant of a rule, then we delete the removed head constraints from the state and add the rule body constraints to it. Any state that is equivalent to this target state is in the transition relation.

The abstract semantics does not account for termination of inconsistent states and propagation rules. From a state with inconsistent built-in constraints, any transition is possible. If a state can fire a propagation rule once, it can do so again and again. This is called trivial non-termination of propagation rules.

Minimum Example, contd. Here is a possible transition from a state $S = (min(0) \wedge min(2) \wedge min(1))$ to a state $T = (min(0) \wedge min(1))$:

$$\frac{\begin{array}{c} S \equiv (min(X) \wedge min(Y) \wedge X \leq Y \wedge (X = 0 \wedge Y = 2 \wedge min(1))) \\ (min(X) \backslash min(Y) \Leftrightarrow X \leq Y | true) \\ (min(X) \wedge X \leq Y \wedge true \wedge (X = 0 \wedge Y = 2 \wedge min(1))) \equiv T \end{array}}{S \mapsto T}$$

2.3 Refined 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.

[1] A variant (renaming) of an expression is obtained by uniformly replacing its variables by fresh variables.

Active and Passive Constraints. As in Prolog, almost all sequential CHR implementations execute queries and rule body constraints 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 [DSGH04]: Every computation in the refined semantics has a corresponding derivation in the abstract semantics, but the converse does not hold. 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.

We consider such a constraint to be *active*. 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. 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 may now hold. Note that the order in which constraints are woken is not fixed in the refined semantics. This allows for optimized data structures for the constraint store that support indexing.

Indexing in CHR. For optimal time complexity, (near) constant-time addition and deletion of CHR constraints is required. Moreover, finding all constraints with a particular value in a particular argument position should be constant-time. To achieve this efficiency, CHR implementations typically provide for indexing on arguments of constraints. Most current CHR libraries in Prolog are based on the KU Leuven CHR system [SD04]. This generic CHR implementation supports indexing for terms via attributed variables based on the principles laid out in [VW10]. In SWI Prolog that we used for implementation, the CHR library also supports hash tables for ground terms and arrays for dense integers. The hash table based indexes in SWI Prolog work at the argument level. For efficient constraint lookups, these arguments have to be ground during computation.

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

We present a conservative extension of CHR by justifications. If they are not used, programs behave as without them. Justifications annotate atomic CHR constraints. A simple source-to-source transformation extends the rules with justifications. We restrict given CHR programs to rules without built-in constraints in the body except *true* and *false*. This restriction is necessary as long as built-in constraint solvers do not support the removal of built-in constraints.

Definition 4 (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* and *rem* (remove) be to unary *reserved* CHR constraint symbols. This means they are only allowed to occur in rules as specified in the following.

Definition 5 (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. More precisely, each CHR constraint in the body is annotated with the union of all justifications in the head of the rule, because its creation is caused by the head constraints. The reserved CHR constraint *rem/1* (remember removed) stores the constraints removed by the rule together with their justifications.

Translating the minimum rule from the introduction to one with justifications results in:

$$min(A)^{F_1} \setminus min(C)^{F_2} \Leftrightarrow A < C \mid F = F_1 \cup F_2 \wedge rem(min(C)^{F_2})^F.$$

3.1 Operational Equivalence of Rule Applications

Let $A, B, C \ldots$ be states. For convenience, we will often consider them as multisets of atomic constraints. Then the notation $A - B$ denotes multiset difference, A without B. To avoid clutter, let $A^{\mathcal{J}}, B^{\mathcal{J}}, C^{\mathcal{J}} \ldots$ stand for 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 a conjunction $\bigwedge_{j=1}^{m} rem(R_j^{F_j})^F$.

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

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

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

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

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

Proof. We compare the two transitions involving rule r and rf, respectively:

$$\frac{(r : H_1 \backslash H_2 \Leftrightarrow C | B) \quad S \equiv (H_1 \wedge H_2 \wedge C \wedge G) \quad (H_1 \wedge C \wedge B \wedge G) \equiv T}{S \mapsto_r T}$$

$$\frac{(rf : H_1^{\mathcal{J}} \backslash H_2^{\mathcal{J}} \Leftrightarrow C | rem(H_2)^{\mathcal{J}} \wedge B^{\mathcal{J}}) \quad S^{\mathcal{J}} \equiv (H_1^{\mathcal{J}} \wedge H_2^{\mathcal{J}} \wedge C \wedge G^{\mathcal{J}}) \quad (H_1^{\mathcal{J}} \wedge C \wedge B^{\mathcal{J}} \wedge G^{\mathcal{J}}) \equiv T^{\mathcal{J}} \wedge rem(H_2)^{\mathcal{J}}}{S^{\mathcal{J}} \mapsto_{rf} T^{\mathcal{J}} \wedge rem(H_2)^{\mathcal{J}}}$$

Given the standard transition with rule r, the transition with justifications with rule rf is always possible: The rule rf by definition does not impose any constraints on its justifications. The justifications in the rule body are computed as the union of the justifications in the rule head, which is always possible. Furthermore, the reserved *rem* constraints always belong to the context of the transition (cf. Definition 3) since by definition there is no rule rf that can match any of them.

Conversely, given the transition with justifications with rule rf, by the same arguments, we can strip away[2] all justifications from it and remove $rem(H_2)^{\mathcal{J}}$ from the rule and the target state to arrive at the standard transition with rule r. □

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.

4 Logical Retraction Using Justifications

We use justifications to remove 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. 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 $kill(f)$ undoes all consequences of the constraint with justification f.

Definition 6 (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.

Distinct variables in the arguments of c are necessary to ensure that the rule matches any instance of a c constraint, independent of its actual arguments. Note that a constraint may be revived and subsequently killed. This is the case when both F_c and F contain the justification f.

[2] For a related strip function, see the proof of Theorem 3.

4.1 Confluence of Logical Retraction

Confluence of a program guarantees that any computation starting from a given initial state can always reach equivalent states, no matter which of the applicable rules are applied. There is a decidable, sufficient and necessary syntactic condition to check confluence of CHR programs and to detect rule pairs that lead to non-confluence when applied.

Definition 7 (Confluence, Joinability). Let A, B and C be states. If $A \mapsto^* B$ and $A \mapsto^* C$ then B and C are joinable.

Two states B and C are joinable if there exist states D_1 and D_2 such that $B \mapsto^* D_1$ and $C \mapsto^* D_2$ where $D_1 \equiv D_2$.

Theorem 1. *[Abd97,AFM99]* A terminating CHR program is confluent if and only if all its critical pairs are joinable.

Decidability comes from the fact that there is only a finite number of critical pairs to consider.

Definition 8 (Overlap, Critical Pair). Given two (not necessarily different) simpagation rules whose variables have been renamed apart, $K_1 \backslash R_1 \Leftrightarrow C_1 | B_1$ and $K_2 \backslash R_2 \Leftrightarrow C_2 | B_2$. Let A_1 and A_2 be non-empty conjunctions of constraints taken from $K_1 \wedge R_1$ and $K_2 \wedge R_2$, respectively. An *overlap* of the two rules is the state consisting of the rules heads and guards:

$$((K_1 \wedge R_1) - A_1) \wedge K_2 \wedge R_2 \wedge A_1 = A_2 \wedge C_1 \wedge C_2.$$

The *critical pair* are the two states that come from applying the two rules to the overlap, where $E = (A_1 = A_2 \wedge C_1 \wedge C_2)$:

$$(((K_1 \wedge K_2 \wedge R_2) - A_2) \wedge B_1 \wedge E <> ((K_1 \wedge R_1 \wedge K_2) - A_1) \wedge B_2 \wedge E).$$

Note that the two states in the critical pair differ by $R_2 \wedge B_1$ and $R_1 \wedge B_2$.

A critical pair is *trivially joinable* if its built-in constraints are inconsistent or if both A_1 and A_2 do not contain removed constraints [AFM99].

We are ready to show the confluence of the *kill* and *revive* rules with each other and with each rule in any given CHR program. It is not necessary that the given program is confluent. This means for any given program, the order between applying applicable rules from the program and retracting constraints can be freely interchanged. It does not matter for the result, if we kill a constraint first or if we apply a rule to it and kill it and its consequences later.

Theorem 2 (Confluence of Logical Retraction). Given a CHR program whose rules are translated to rules with justifications together with the *kill* and *revive* rules. We assume there is at most one $kill(f)$ constraint for each justification f in any state. Then all critical pairs between the *kill* and *revive* rules and any rule from the program with justifications are joinable.

Proof. There are several overlaps between various rules to consider.

There is only one overlap between the *kill* and *revive* rules,

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

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

since G^F cannot have the reserved constraint symbol $\mathit{rem}/1$. The overlap is in the $\mathit{kill}(f)$ constraint. But since it is not removed by any rule, the resulting critical pair is trivially joinable.

By our assumption, the only overlap between two instances of the *kill* rule must have a single $\mathit{kill}(f)$ constraint. Again, since it is not removed, the resulting critical pair is trivially joinable. The same argument applies to the only overlap between two instances of the *revive* rule.

Since the head of a simpagation rule with justifications from the given program

$$rf : K^{\mathcal{J}} \setminus R^{\mathcal{J}} \Leftrightarrow C \mid \mathit{rem}(R)^{\mathcal{J}} \wedge B^{\mathcal{J}}$$

cannot contain reserved *kill* and *rem* constraints, these program rules cannot have an overlap with the *revive* rule.

But there are overlaps between program rules, say a rule rf, and the *kill* rule. They take the general form:

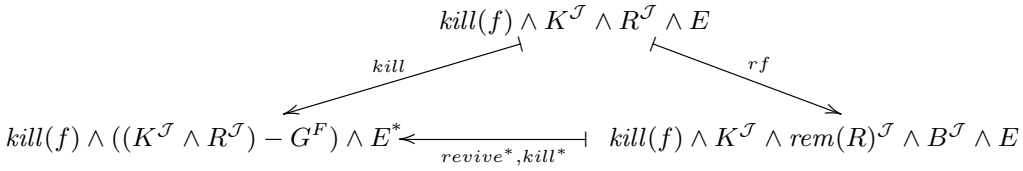
$$\mathit{kill}(f) \wedge K^{\mathcal{J}} \wedge R^{\mathcal{J}} \wedge G^F{=}A^F \wedge f{\in}F \wedge C,$$

where A^F occurs in $K^{\mathcal{J}} \wedge R^{\mathcal{J}}$. This leads to the critical pair

$$(\mathit{kill}(f) \wedge ((K^{\mathcal{J}} \wedge R^{\mathcal{J}}) - G^F) \wedge E <> \mathit{kill}(f) \wedge K^{\mathcal{J}} \wedge \mathit{rem}(R)^{\mathcal{J}} \wedge B^{\mathcal{J}} \wedge E),$$

where $E = (G^F{=}A^F \wedge f{\in}F \wedge C)$. In the first state of the critical pair, the *kill* rule has been applied and in the second state the rule rf. Note that A^F is atomic since it is equated to G^F in E. Since G^F has been removed in the first state and $G^F{=}A^F$, rule rf is no longer applicable in that state.

We would like to join these two states. The joinability between a rule rf and the *kill* rule can be visualized by the diagram:



We now explain this joinability result. The states of the critical pair differ. In the first state we have the constraints $R^{\mathcal{J}}$ and have G^F removed from $K^{\mathcal{J}} \wedge R^{\mathcal{J}}$, while in the second state we have the body constraints $\mathit{rem}(R)^{\mathcal{J}} \wedge B^{\mathcal{J}}$ of rule rf instead. Any constraint in $\mathit{rem}(R)^{\mathcal{J}} \wedge B^{\mathcal{J}}$ must include f as justification by definition, because f occurred in the head constraint A^F and E contains $f{\in}F$.

The goal $\mathit{rem}(R)^{\mathcal{J}}$ contains *rem* constraints for each removed constraint from $R^{\mathcal{J}}$. But then we can use $\mathit{kill}(f)$ with the *revive* rule to replace all *rem* constraints

by the removed constraints, thus adding $R^{\mathcal{J}}$ back again. Furthermore, we can use $kill(f)$ with the *revive* rule to remove each constraint in $B^{\mathcal{J}}$, as each constraint in $B^{\mathcal{J}}$ contains the justification f. So $rem(R)^{\mathcal{J}} \wedge B^{\mathcal{J}}$ has been removed completely and $R^{\mathcal{J}}$ has been re-added.

The two states may still differ in the occurrence of G^F (which is A^F). In the first state, G^F was removed by the *kill* rule. Now if A^F (G^F) was in $R^{\mathcal{J}}$, it has been revived with $R^{\mathcal{J}}$. But then the *kill* rule is applicable and we can remove A^F again. In the second state, if A^F was in $R^{\mathcal{J}}$ it has been removed together with $R^{\mathcal{J}}$ by application of rule *rf*. Otherwise, A^F is still contained in $K^{\mathcal{J}}$. But then the *kill* rule is applicable to A^F and removes it from $K^{\mathcal{J}}$. Now A^F (G^F) does not occur in the second state either.

We thus have arrived at the first state of the critical pair. Therefore the critical pair is joinable. □

This means that given a state, if there is a constraint to be retracted, we can either kill it immediately or still apply a rule to it and use the *kill* and *revive* rules afterwards to arrive at the same resulting state.

Note that the confluence between the *kill* and *revive* rules and any rule from the program is independent of the confluence of the rules in the given program.

4.2 Undoing of Rule Applications

From the proof of confluence in Theorem 2 we can see that killing a constraint also undoes the rule application it was involved in. We can show that one can undo the rule application, provided we do not kill the associated initial constraint. The following Lemma appears here for the first time.

Lemma 2 (Rule Undoing). Given a state $A^{\mathcal{J}} \wedge G^{\{f\}} \wedge kill(f)$ where f does not occur in $A^{\mathcal{J}}$. Then any application of a rule that involves $G^{\{f\}}$ can be undone by applications of the *kill* and *revive* rules for $kill(f)$:

$$\text{If} \quad A^{\mathcal{J}} \wedge G^{\{f\}} \wedge kill(f) \mapsto_{rf} D^{\mathcal{J}} \wedge kill(f)$$

$$\text{then} \quad D^{\mathcal{J}} \wedge kill(f) \mapsto^*_{kill,revive} A^{\mathcal{J}} \wedge G^{\{f\}} \wedge kill(f)$$

Proof. The structure of the proof is very similar to that of the proof of confluence in Theorem 2.

Given a simpagation rule with justifications

$$rf : K^F \backslash R^F \Leftrightarrow C | rem(R)^F \wedge B^F$$

Then the transition that applies the rule *rf* is as follows:

$$A^{\mathcal{J}} \wedge G^{\{f\}} \wedge kill(f) \mapsto_{rf} D^{\mathcal{J}} \wedge kill(f)$$

where $A^{\mathcal{J}} \wedge G^{\{f\}} \equiv K^F \wedge R^F \wedge C \wedge E$ and $D^{\mathcal{J}} \equiv K^F \wedge rem(R)^F \wedge B^F \wedge C \wedge E$.

We also have that $G^{\{f\}}$ occurs in $K^F \wedge R^F$ in the source state $K^F \wedge R^F \wedge C \wedge E$. The justification f occurs in all constraints of $rem(R)^F \wedge B^F$ in the

target state $K^F \wedge rem(R)^F \wedge B^F \wedge C \wedge E$, because the rule application involved $G^{\{f\}}$. Exhaustive application of the *revive* rule to $rem(R)^F$ will remove these constraints and re-add R^F. Exhaustive application of the *kill* rule to B^F will remove these constraints. Therefore the resulting state is $K^F \wedge R^F \wedge C \wedge E$, i.e. we are back to the source state. □

4.3 Correctness of Logical Retraction

We prove 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 show 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 are not applicable, i.e. these rules have been applied to exhaustion, then there is a corresponding computation without constraints that contain the justification f.

Theorem 3 (Correctness of Logical Retraction). 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 $kill(f)$

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

Proof. We distinguish between transitions that involve the justification f or do not. A rule that applies to constraints that do not contain the justification f will produce constraints that do not contain the justification. A rule application that involves at least one constraint with a justification f will only produce constraints that contain the justification f.

We now define a mapping from a computation with $G^{\{f\}}$ to a corresponding computation without $G^{\{f\}}$. The mapping essentially strips away constraints that contain the justification f except those that are remembered by *rem* constraints. In this way, the exhaustive application of the *revive* and *kill* rules $kill(f)$ is mimicked.

$strip(f, A^{\mathcal{J}} \wedge B^{\mathcal{J}}) := \ strip(f, A^{\mathcal{J}}) \wedge strip(f, B^{\mathcal{J}})$
$strip(f, rem(G^{F_1})^{F_2}) := \ strip(f, G^{F_1})$ if $f \in F_2$
$strip(f, G^F) := \ true$ if G is an atomic constraint except *rem/1* and $f \in F$
$strip(f, G^F) := \ G^F$ otherwise.

We extend the mapping from states to transitions. We keep the transitions except where the source and target state are equivalent, in that case we replace the transition $\mapsto$ by an equivalence $\equiv$. This happens when a rule is applied that involves the justification f. The mapping is defined in such a way that in this case the source and target state are equivalent. Otherwise a rule that does not involve f has been applied. The mapping ensures in this case that all necessary constraints are in the source and target state, since it keeps all constraints that

do not mention the justification f. For a computation step $C^{\mathcal{J}} \mapsto D^{\mathcal{J}}$ we define the mapping as:

$$\begin{aligned} \mathit{strip}(f, C^{\mathcal{J}} \mapsto_{rf} D^{\mathcal{J}}) &:= \ \mathit{strip}(f, C^{\mathcal{J}}) \equiv \mathit{strip}(f, D^{\mathcal{J}}) \text{ if rule } \mathit{rf} \text{ involves } f \\ \mathit{strip}(f, C^{\mathcal{J}} \mapsto_{rf} D^{\mathcal{J}}) &:= \ \mathit{strip}(f, C^{\mathcal{J}}) \mapsto_{rf} \mathit{strip}(f, D^{\mathcal{J}}) \text{ otherwise.} \end{aligned}$$

We next have to show is that the mapping results in correct state equivalences and transitions. If a rule is applied that does not involve justification f, then it is easy to see that the mapping *strip(f, ...)* leaves states and transitions unchanged.

Otherwise the transition is the application of a rule *rf* from the program, the rule *kill* or the rule *revive* where f is contained in the justifications. Let the context $E^{\mathcal{J}}$ be an arbitrary goal where $f \in \mathcal{J}$. Then we have to compute

$$\mathit{strip}(f, \mathit{kill}(f) \wedge G^F \wedge f \in F \wedge E^{\mathcal{J}} \mapsto_{\mathit{kill}} \mathit{kill}(f) \wedge E^{\mathcal{J}})$$

$$\mathit{strip}(f, \mathit{kill}(f) \wedge \ \mathit{rem}(G^{F_c})^F \wedge f \in F \wedge E^{\mathcal{J}} \mapsto_{\mathit{revive}} \mathit{kill}(f) \wedge G^{F_c} \wedge E^{\mathcal{J}})$$

$$\mathit{strip}(f, K^{\mathcal{J}} \wedge R^{\mathcal{J}} \wedge C \wedge E^{\mathcal{J}} \mapsto_{\mathit{rf}} K^{\mathcal{J}} \wedge \mathit{rem}(R)^{\mathcal{J}} \wedge B^{\mathcal{J}} \wedge C \wedge E^{\mathcal{J}})$$

and to show that equivalent states are produced in each case. The resulting states are

$$\begin{gathered} \mathit{true} \wedge \mathit{true} \wedge \mathit{true} \wedge E^{\mathcal{J}'} \equiv \mathit{true} \wedge E^{\mathcal{J}'} \\ \mathit{true} \wedge G^{F_c} \wedge \mathit{true} \wedge E^{\mathcal{J}'} \equiv \mathit{true} \wedge G^{F_c} \wedge E^{\mathcal{J}'} \text{ if } f \notin F_c \\ \mathit{true} \wedge \mathit{true} \wedge \mathit{true} \wedge E^{\mathcal{J}'} \equiv \mathit{true} \wedge \mathit{true} \wedge E^{\mathcal{J}'} \text{ if } f \in F_c \\ K^{\mathcal{J}'} \wedge R^{\mathcal{J}'} \wedge C \wedge E^{\mathcal{J}'} \equiv K^{\mathcal{J}'} \wedge R^{\mathcal{J}'} \wedge C \wedge E^{\mathcal{J}'} \text{ where } f \notin \mathcal{J}', \end{gathered}$$

where, given a goal A, the expression $A^{\mathcal{J}'}$ contains all constraints from $A^{\mathcal{J}}$ that do not contain the justification f.

In the end state of the given computation we know that the *revive* and *kill* rules have been applied to exhaustion. Therefore all $\mathit{rem}(G^{F_1})^{F_2}$ where F_2 contains f have been replaced by G^{F_1} by the *revive* rule. Therefore all standard constraints with justification f have been removed by the *kill* rule (including those revived), just as we do in the mapping *strip(f, ...)*. The end states are indeed equivalent except for the remaining *kill* constraint. □

4.4 Strong Correctness for Confluent Programs

For confluent programs, Theorem 3 can be tightened.

Theorem 4 (Strong Correctness of Logical Retraction). Given a confluent CHR program P, its translation to a program with justifications $P^{\mathcal{J}}$ and a computation in $P^{\mathcal{J}}$

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

where f does not occur in $A^{\mathcal{J}}$. Then for *every* computation

$$A^{\mathcal{J}} \mapsto^* B^{\mathcal{J}} \wedge \mathit{rem}(R)^{\mathcal{J}} \not\mapsto .$$

Proof. We first show that if the given CHR program P is confluent, so is its translation $P^{\mathcal{J}}$. If P is confluent, so is the translation of its rules to rules with justifications by Lemma 1. So all critical pairs are joinable. For $P^{\mathcal{J}}$, we also have to add the *kill* and *revive* rules. By Theorem 2, all critical pairs between these rules and rules from $P^{\mathcal{J}}$ are joinable. Hence all critical pairs between any pair of rules in $P^{\mathcal{J}}$ are joinable, therefore $P^{\mathcal{J}}$ is confluent.

We now show the strong correspondence between the computations with and without $G^{\{f\}}$ and $kill(f)$. The states $B^{\mathcal{J}} \wedge rem(R)^{\mathcal{J}} \wedge kill(f)$ and $B^{\mathcal{J}} \wedge rem(R)^{\mathcal{J}}$, respectively, admit no further computation steps, they are final states. As $P^{\mathcal{J}}$ is confluent, by definition any computation from a given initial state can always reach equivalent states. This implies that for each initial state, there is a unique final state in a terminating computation.

By Theorem 1, for any state

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

there exists a computation

$$A^{\mathcal{J}} \mapsto^* C^{\mathcal{J}}.$$

By confluence we know that

$$C^{\mathcal{J}} \wedge kill(f) \mapsto^* B^{\mathcal{J}} \wedge rem(R)^{\mathcal{J}} \wedge kill(f) \not\mapsto .$$

In this computation, the constraint $kill(f)$ can never be active. Only the rules *kill* and *revive* could apply to a $kill(f)$ constraint. Initially $kill(f)$ is not active, because Theorem 1 presumes $C^{\mathcal{J}} \wedge kill(f) \not\mapsto_{kill,revive}$, where f does not occur in $A^{\mathcal{J}}$. In the remaining computation, f cannot be reintroduced in another constraint, because program rules do not apply to $kill(f)$. Therefore the rules *kill* and *revive* are never applicable with $kill(f)$ and $kill(f)$ can never become active.

Hence for any transition, $kill(f)$ will be in the context of the transition (cf. Definition 3) and cannot impede the applicability of a rule. So the computation

$$C^{\mathcal{J}} \wedge kill(f) \mapsto^* B^{\mathcal{J}} \wedge rem(R)^{\mathcal{J}} \wedge kill(f) \not\mapsto$$

has a corresponding computation

$$C^{\mathcal{J}} \mapsto^* B^{\mathcal{J}} \wedge rem(R)^{\mathcal{J}} \not\mapsto$$

that only differs in that the constraint $kill(f)$ is removed from every state.

From the above computations

$$A^{\mathcal{J}} \mapsto^* C^{\mathcal{J}} \text{ and } C^{\mathcal{J}} \mapsto^* B^{\mathcal{J}} \wedge rem(R)^{\mathcal{J}} \not\mapsto$$

it follows that

$$A^{\mathcal{J}} \mapsto^* B^{\mathcal{J}} \wedge rem(R)^{\mathcal{J}} \not\mapsto .$$

Since the program is confluent and since $B^{\mathcal{J}} \wedge rem(R)^{\mathcal{J}}$ is a final state, it is the only possible final state for $A^{\mathcal{J}}$. □

5 Basic Implementation

As a proof-of-concept, we implement CHR with justifications (CHR$^{\mathcal{J}}$) in SWI-Prolog using its CHR library. This prototype source-to-source transformation is available online at `http://pmx.informatik.uni-ulm.de/chr/translator/`. The translated programs can be run in Prolog or online systems like WebCHR. As we will clarify in this section, for correctness of the implementation we have to assume confluence of the given CHR program.

5.1 Direct Implementation

We give a straightforward direct implementation of constraints and rules extended by justifications.

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 represented 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

```
adjust @ 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 translated as follows[3]:

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

```
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.

Rules `remove` and `revive`. Justifications are represented 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`, where `r` is an arbitrary unique constant. In this way the justification variable gets bound, which is interpreted as setting the corresponding flag. If $kill(f)$ occurred in the head of a *kill* or *revive* rule, it is moved to the guard as equality test `F==r`. Note that we rename rule *kill* to `remove` in the implementation.

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

[3] 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.

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

These rules are added before any other rule of the transformed program. Since rules are tried in program order in the CHR implementation, the constraint `C` in the second rule is not a reserved `rem/1` constraint when the rule is applicable. 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 one occurrence of a constraint `C`. The two rules `killc` and `killr` try to find the constraint `C`. The `killr` rule applies in the case where constraint `C` has been removed and is therefore 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 which is implemented by the `member` predicate.

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

Note that in the `killr` 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.

5.2 Correctness

Since the rules in the implementation directly correspond to the abstract rules given before, their correctness is easy to see. However this is not sufficient to show correctness of the overall implementation. The proofs of our theorems we gave are in the abstract semantics, the implementation is in the refined semantics.

The refined semantics is a proven refinement of the abstract one [DSGH04] essentially fixing the order of rule applications and constraint activations. Rules are tried in the order in which they occur in the program. Constraints become active in the order in which the occur in a state.

Every computation in the refined semantics has a corresponding derivation in the abstract semantics [DSGH04], but the converse does not hold. So we may miss computations in the implementation that are possible in the abstract semantics and that are necessary for the theorems of the previous section to hold.

Counter-Example for Correctness. Consider the following program (we use the untranslated code for readability):

```
a, b <=> e.
a, c <=> f.
a, d <=> g.
```

and the query

```
?- a, b, d, c.
```

In the refined semantics, the only possible computation leads to

```
e, d, c.
```

while in the abstract semantics, any of the three program rules could be applied. But this is not yet the problem. The problem is exemplified by the following computation

```
?- a, d, c.
   g, c.
```

compared to

```
?- a, b, d, c, killc(b).
   f, d.
```

The difference in the final states is incorrect but occurs in the refined semantics: constraints are activated from left to right and they are delayed (put into the constraint store) when no rule is applicable to them. When `killc(b)` is reached and becomes active, the constraints `e, d, c` are in the constraint store. Then `e` will be killed, `b` will be revived and killed, and `a` will be revived, `d, c` stay in the store. So `a` is now active. Since rules are applied in the order in which they occur in the program, `a` will apply the rule `a, c <=> f` which leads to the final state given above.

Confluence for Correctness. The example program above is not confluent. Confluence solves the problem, because it guarantees that any computation from a given initial state can always reach equivalent states independent of the order of constraint activations and rule applications. This implies that final states are equivalent.

We now show that Theorem 4 which was formulated for the abstract semantics also holds for the refined semantics for terminating programs and then applies to our implementations.

In [DSGH04] it is proven in Corollary 1 that if a CHR program is terminating under the refined semantics and confluent under the abstract semantics, it is also confluent under the refined semantics.

Since any computation in the refined semantics is also possible in the abstract semantics [DSGH04], the final states of any computation from a given initial state must be equivalent in both semantics if the program is confluent in both semantics.

For the correctness of our implementations, it is therefore a sufficient condition that the given CHR program to be extended with justifications is terminating and confluent.

5.3 Examples

We discuss two classical examples for dynamic algorithms, maintaining the minimum of a changing set of numbers and shortest paths when edges change.

Dynamic Minimum. 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]`.

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

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].
```

On the other hand, if we retract the minimum `min(0)`, the `killc` rule

```
killc @  killc(C), C ## Fs <=> member(F,Fs),F=r
```

applies. It removes `min(0)##[B]` and binds justification `B`. The two `rem` constraints for `min(1)` and `min(2)` have justification `B` as well, so these two constraints are re-introduced by applications of rule `revive`

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

The minimum rule applies to these two revived constraints. Note that `min(2)` is now removed by `min(1)` (before it was `min(0)`). The result is the updated minimum, which of course is `1`:

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

Dynamic Shortest Path. Given a graph with directed arcs `e(X,Y)`, we compute the lengths of the shortest paths between all pairs of reachable nodes:

```
 % keep shorter of two paths from X to Y
pp @ p(X,Y,L1) \ p(X,Y,L2) <=> L1=<L2 | true.
 % edges have a path of unit length
e  @ e(X,Y) ==> p(X,Y,1).
 % extend path in front by an edge
ep @ e(X,Y), p(Y,Z,L) ==> L1=:=L+1 | p(X,Z,L1).
```

The corresponding rules in the translated program are:

```
pp@p(A,B,C)##D \ p(A,B,E)##F <=> C=<E |
                                union([D,F],G), rem(p(A,B,E)##F)##G.
e @e(A,B)##C ==> true | union([C],D), p(A,B,1)##D.
ep@e(A,B)##C,p(B,D,E)##F ==> G is E+1 | union([C,F],H),p(A,D,G)##H.
```

Here is a sample query and its resulting answer.

```
?- e(a,b)##[A], e(b,c)##[B], e(a,c)##[C].
rem(p(a, c, 2)##[A, B])##[A,B,C],
p(a, b, 1)##[A], e(a, b)##[A],
p(b, c, 1)##[B], e(b, c)##[B],
p(a, c, 1)##[C], e(a, c)##[C].
```

We see that a path of length 2 has been removed by the constraint `e(a,c)##[C]`, which produced a shorter path of length one. We next kill this constraint `e(a,c)`.

```
?- e(a,b)##[A], e(b,c)##[B], e(a,c)##[C], kill(e(a,c)).
p(a, b, 1)##[A], e(a, b)##[A],
p(b, c, 1)##[B], e(b, c)##[B],
p(a, c, 2)##[A,B].
```

Its path `p(a,c,1)` disappears and the removed path `p(a,c,2)` is re-added. We can see that the justifications of a path contains are those from the edges in that path. The same happens if we logically retract `p(a,c,1)` instead of `e(a,c)`.

What happens if we remove `p(a,c,2)` from the initial query? The `killr` rule applies. Since the path has two justifications, there are two computations generated by the `member` predicate. In the first one, the constraint `e(a,b)` disappeared, in the second answer, it is `e(b,c)`. In both cases, the path cannot be computed anymore, i.e. it has been logically retracted.

```
?- e(a,b)##[A], e(b,c)##[B], e(a,c)##[C], kill(p(a,c,2)).
p(b, c, 1)##[B], e(b, c)##[B],
p(a, c, 1)##[C], e(a, c)##[C]
;
p(a, b, 1)##[A], e(a, b)##[A],
p(a, c, 1)##[C], e(a, c)##[C].
```

6 Optimizing the Implementation

We describe in this section the modifications for our improved implementations following [Fru17a] and in the next section for the first time we formally show the space and time complexity of this 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. However, in the basic implementation, the computation time of a union of justifications is linear in the size of its input

justification sets. So it depends on the number of initial justifications, i.e. the number of constraints in the given query. A first idea to avoid this cost is to delay this computation until it is needed due to a retraction. But we can actually go farther in our constraint-based setting. We actually need never compute the union of justifications, but will use the `union` constraints as data to find the necessary justifications.

6.1 Optimized Implementation

To logically retract a constraint with justification `F`, we will use the constraints `killj(F), killone`. The constraint `killj(F)` (kill justification) finds the initial justifications from which justification `F` derived. The constraint `killone` will then choose one initial justification for logical retraction. We first modify the rules for the `killc` constraint accordingly.

```
killc @  killc(C), C ## Fs <=> killj(Fs), killone.
killr @  killc(C), rem(C ## FC) ## _Fs <=> killj(FC), killone.
```

The computation then proceeds in four phases as follows.

Phase 1 - Finding all initial justifications with `killj`. Since the delaying `union` constraints are inactive data now, their arguments (which are justifications) are unbound variables, except for occurrences of initial justifications, which are represented as singleton lists.

The constraint `killj` 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 will proceed recursively with the help of `killl` (kill list) until it reaches an initial justification. We can stop if we see a justification again that we have already seen using rule `already_seen`.

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

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

If no computation caused by `killj(F)` is possible anymore, there is exactly one `killj` constraint for every justification from which justification `F` derived.

Phase 2 - Choosing an initial justifications with `killone`. Then the auxiliary constraint `killone` (kill one) becomes active and chooses one of the initial justifications and removes it.

```
killit @ killone, killj([F]) <=> F=r, dorev.
```

In rule `killit`, we recognize an initial justification by its form `[F]`. The binding of variable `F` to constant `r` marks it as to be killed and wakes up all constraints in which this justification occurs. By this mechanism, the constraints can be retracted and revived. The auxiliary constraint `dorev` (do revive) will be used to delay the reviving of constraints. Note that the rule is applied exactly one.

Phase 3 - Killing justifications and removing constraints with `F=[r]`. Once an initial justification was chosen for retraction, we also have to kill all output justifications of unions that have a killed justification as input justification, i.e. we go forward using the `union` constraints[4].

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

We remove constraints with killed justifications.

```
remove @ c(X1,..Xn,[r]) <=> true.
```

Note that 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.

Phase 4 - Reviving constraints with `dorev`. Only when all constraints have been removed, constraints are revived. This completes the logical retraction and improves the performance of the subsequent partial recomputation. The constraint `dorev` now triggers the re-addition of previously removed constraints.

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

When all constraints have been revived, `dorev` is removed at the very end by the last rule.

6.2 Correctness

To show correctness of the optimized implementation, we establish its equivalence with the basic implementation which we already have shown correct.

Rewriting the optimized program. We first rewrite the rules of the optimized implementation without changing its semantics. In this way we make the relation to the basic version more explicit. Every built-in constraint `F=[r]` in the body of a rule is replaced by the built-in constraints `member(F2,F),F2=r`, where `F2` is a new variable. Likewise, every built-in constraint `F==r` in the guard of a rule is replaced by the built-in constraints `member(F2,F),F2==r`, and every occurrence of `[r]` in the head of a rule is replaced by a new variable[5] `F`, and `member(F2,F),F2==r` is added to the guard of the rule[6]. Moreover, `c(X1,..Xn,F)` is replaced by `c(X1,..Xn)##F`. This results in the following rewritten optimized program:

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

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

[4] To avoid special cases, in the rule we use the term `[r]` as for initial justifications instead of just `r`.

[5] For initial justifications, `F` becomes `[F]`.

[6] We assume in this subsection that the built-in `member/2` is deterministic, meaning that it will just delay if its second argument is a variable.

```
killit @ killone, killj([F]) <=> member(F2,[F]),F2=r, dorev.

go_forward @  union(FL,F) <=> member(F1,FL), member(F2,F1),F2==r |
                                  member(F3,F),F3=r.

remove @ c(X1,..Xn)##F <=> member(F2,F),F2==r | true.
revive @ dorev \ rem(c(X1,..Xn)##FC)##F <=> member(F2,F),F2==r |
                                  c(X1,..Xn,FC).
clean_up @ dorev <=> true.
```

When establishing the relationship to the basic version, it is easier to consider the rules of the programs in backward order from the last to the first.

Correctness of the `remove` and `revive` rules. The rules `remove` and `revive` are now identical to the ones in the basic version, except that `dorev` must be present in the `revive` rule. What is left to show for correctness is that the remaining rules lead to the same behavior as when `union` is evaluated in the basic version and that a `dorev` constraint will be produced.

Correct meaning of the remaining rules. The remaining rules actually express properties of `union/2` with regard to justifications in its list arguments.

The rule `go_forward` says that if an element of a list in the input list `FL` is bound to `r`, then it is also bound in the output list `F`.

Recall that logical retraction is expressed by the initial goal `killj(F0), killone`. The variable `F0` is now interpreted as list of justifications. Any constraint `killj(F)` occurring in the computation of the initial goal then means that the list `F` only contains initial justifications from the list `F0`.

The rule `killit` says that if `killj([F])` contains only justifications from `F0` and it is an initial justification, then we can bind (and thus kill) it.

The rules `go_to_intial` and `killl` say that for each `union(FL,F)`, if `F` contains only justifications from `F0`, then all lists contained in the list `FL` do so as well.

Finally, the rule `already_seen` removes duplicates of `killj` constraints.

Correct execution of the remaining rules. So far we have shown that the meaning of the rules is correct. We have not shown yet that the execution of these rules are sufficient to replace the evaluation of the `union` constraint in the basic version of our program. We now proceed with the rules in the order as given in the program, i.e. in the order in which they are executed in the refined semantics.

In the basic version, we have direct access to the list of initial justifications for a given constraint. The rules for `killc/1` will then choose one of them. In the optimized version, we have to find all initial justifications first by going through `union` constraints starting from `killj(F0), killone`.

By the definition of `union/2`, all justifications in the first argument occur in the second argument and vice versa. If there is a constraint `union(FL,F0)`, then `F0` is an unbound variable in our optimized version, but we know that its justifications are exactly those of `FL`. This kind of reasoning is made executable by the rules `go_to_intial` and `killl`. Their correctness is easy to be seen due

to their minimality. It is also clear removing duplicates in rule `already_seen` does not impede correctness, but improves performance by avoiding redundant computations with duplicate `killj` constraints.

When an active `killj(F)` constraint cannot find a partner constraint `union` anymore, we know that `F` must contain a single initial justification. When the rules for `killj` have been applied to exhaustion, all initial justification have been found. Thus the rules involving `killj` in Phase 1 of the optimized version choose and bind an initial justification as do the rules for `killc` in the basic version.

In Phase 2 of our optimized implementation, `killone` becomes active and by applying rule `killit` exactly once, it will choose a `killj` constraint with an initial justification, and bind that justification to `r`, which wakes all constraints in which this justification occurs. When the woken constraints are all processed, `dorev` will become active.

The constraints that are woken are `union/2, c/n+1, rem/2` constraints. `rem/2` constraints will delay again, they will be processed when `dorev` with rule `revive` becomes active. `c/n+1` constraints will be removed with rule `remove`. The woken `union/2` constraints will contain the bound justification in the first argument. Therefore the guard of rule `go_forward` holds and the rule can be applied. It will bind the justification in the second argument, which in turn will wake further constraints. Rule `go_forward` will exhaustively bind justifications in all arguments of `union` constraints that involve the chosen initial justification.

In this way all constraints that depend on the initial justification will be removed and revived as in the basic implementation. Therefore Correctness Theorem 4 also holds for the optimized implementation in the refined semantics provided the given program is terminating and confluent.

7 Worst-Case Space and Time Complexity

We are interested in the computational overhead of introducing justifications using our optimized program transformation and of performing logical retraction in a given query. We will see that in both cases, the additional time and space needed are proportional to the derivation length of the query in the worst case. We will also show that this overhead may increase space complexity, but does not change the time complexity with regard to the original untransformed program.

We give upper-bounds for the worst-case space and time complexity of the overhead for justifications by inspecting the rules of our transformed program. Let A be a query (initial state) for a given program $\mathcal{P}$ without justifications. Let $A^{\mathcal{J}}$ be this query and $\mathcal{P}^{\mathcal{J}}$ be this program extended with justifications according to the optimized implementation of the source-to-source-transformation in Section 6. Note that A and thus $A^{\mathcal{J}}$ do not contain constraints for logical retraction (`killc, killj, killone,...`). Let c be the number of CHR constraints in the query A. Let k be the largest number of head or body constraints of a rule in the program $\mathcal{P}$. Note that k is a constant for a given program. Let n be the derivation length of a computation for query A in program $\mathcal{P}$, i.e. the number

of rule applications (transitions). Since a rule application takes at least constant time, the time complexity of the original program cannot be better than $O(n)$.

In the refined semantics, rules are applied in textual order of the program and that queries and body constraints are activated in textual order. We assume that argument-wise indexing is available as described in Section 2.3. We assume that constraints can be added (inserted) in constant time. We assume that if a variable is bound, only the constraints that contain that variable are considered for re-activation (waking). These assumptions hold for most sequential CHR implementations [SF06,DSGH04,VW10].

Recall the rules for handling logical retraction from Section 6:

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

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

killit @ killone, killj([F]) <=> F=r, dorev.

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

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

We start with the following lemma.

Lemma 3 (Constant Time Justification Handling Rule Application). All rule tries (application attempts) and rule applications for handling logical retraction take constant time.

Proof. Constant time of rule application attempts is mostly achieved by indexing on the arguments that contain a justification:

- rule `already_seen` and rule `go_to_initial` use the index on the justification `F`,
- rules `killl` use an index to distinguish between empty and non-empty lists,
- rule `killit` uses an index to find a singleton justification in a list,
- rule `go_forward` involves a single head constraint and has a guard where the `member` predicate has to go through a list of k justifications at most, but k is a constant,
- rule `remove` and rule `revive` use the index on the justification `F`,
- rule `clean_up` involves a single head constraint.

Furthermore, constant time rule applications are possible: The number of constraints in a rule is always bounded and by our assumption body constraints of a rule can be added (inserted) in constant time. □

7.1 Overhead of Justifications

We determine the overhead in terms of space and time for introducing justifications.

Lemma 4 (Space Complexity of Overhead for Justifications). Execution of a query $A^{\mathcal{J}}$ in the transformed program $\mathcal{P}^{\mathcal{J}}$ has an overhead in space proportional to the derivation length n of the query A in the original program $\mathcal{P}$.

Proof. The query $A^{\mathcal{J}}$ is without logical retraction, so only the transformed rules of the original program apply. From Lemma 1 we can conclude that these rules can be applied in the same way as the original rules for query A.

Recall that each rule of the original program is transformed according to the following scheme (see Section 3):

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

Each application of the transformed rule with justifications introduces in addition to the at most k original body constraints one `union` constraint with a list of at most k elements and one new justification `Fs` and at most k `rem` constraints, where k is a constant. The additional rules handling justifications in case of logical retraction will not be applied, so no space is needed for their constraints. Therefore the number of constraints additionally introduced is proportional to the number of rule applications n. □

So in order to compute the space complexity of the query $A^{\mathcal{J}}$ in transformed program $\mathcal{P}^{\mathcal{J}}$, $O(n)$ has to be added to the space complexity of the query A in the original program $\mathcal{P}$.

We next determine the overhead in terms of time for introducing justifications.

Lemma 5 (Time Complexity of Overhead for Justifications). Execution of a query $A^{\mathcal{J}}$ in the transformed program $\mathcal{P}^{\mathcal{J}}$ has an overhead in time proportional to the derivation length n of the query A in the original program $\mathcal{P}$.

Proof. Without logical retraction in query $A^{\mathcal{J}}$, only the transformed rules of the original program apply. From Lemma 1 we can conclude that these rules can be applied with the same derivation length and the same time complexity as the original rules.

The additional rules handling justifications in case of logical retraction[7] will not be applicable, but may be tried by the constraints in the body of the applied program rule. There are at most k CHR constraints: one `union` constraint, `rem` constraints and CHR constraints with justifications. By Lemma 3, these rule application attempts each take constant time. Their number is proportional to n in the worst case. So they do not add to the overall time complexity. □

[7] See the proof of forthcoming Lemma 7 for a related discussion of these rules.

The time complexity of the query $A^{\mathcal{J}}$ in transformed program $\mathcal{P}^{\mathcal{J}}$ is the same as the time complexity of the query A in the original program $\mathcal{P}$. It is at least $O(n)$. The introduction of justifications without logical retraction does not increase the time complexity of the original program.

7.2 Overhead of Logical Retraction

We now consider the complexity of performing logical retraction. We can determine the overhead involved in handling the logical retraction itself. The additional time and space needed for partial recomputation, i.e. for the execution of revived constraints on the other hand depends on the given program. We therefore cannot include it in the overhead considered below. Logical retraction pays off if these additional costs in terms of execution time are less than recomputing the original query from scratch with the retracted initial constraint removed.

To logically retract constraints with justification F, we use the query $A^{\mathcal{J}} \wedge$ `killj`(`F`) $\wedge$ `killone`. The constraint `killj` proceeds recursively through the `union` constraints to find the initial justifications associated with the justification `F`. Then the constraint `killone` chooses one initial justification. Its associated initial constraint from the query is removed and all its consequences (rule applications) are undone by removing and reviving constraints that derive from the initial constraint.

Lemma 6 (Time Complexity of Overhead for Logical Retraction). Execution of a logical retraction with query $A^{\mathcal{J}} \wedge$ `killj`(`F`) $\wedge$ `killone` in the transformed program $\mathcal{P}^{\mathcal{J}}$ has an overhead in time proportional to the derivation length n of the query A in the original program $\mathcal{P}$ in the worst case.

Proof. We first recall the basic workings of the rules for handling logical retraction. The nested recursion of constraints `killj` and `killl` goes along the justification in the `union` constraints with the rules `already_seen`, `go_to_initial` and `killl`. An initial justification is choosen by the constraint `killone` in the rule `killit`. It binds the justification variable. This will wake up all constraints in which the variable occurs. These are all `union` constraints and all `rem` and program constraints that have this justification. Thus the rule `go_forward` and `remove` are immediately applicable, while the `revive` rule applications have to wait for the constraint `dorev`. It is added after the binding of the justification in rule `killit`.

By assumption, rules and constraints are applied in textual order. Thus rule `already_seen` ensures that all `killj` constraints have different justifications as arguments. Any duplicate constraint will be immediately removed. In a query with a derivation of length n, there are exactly n `union` constraints with up to k (a constant) old justifications in the list argument and one new justification, all different from each other. There are c initial justifications (one for each initial constraint in the query). Hence there can only be up to $c + n$ different justifications as well as `killj` constraints. Each of the n rule applications also produces at most k `rem` constraints and k body constraints with justifications.

Thus there are at most kn such constraints. Finally, there will be one `killone` and one `dorev` constraint.

By Lemma 3 we already know that all rule tries (application attempts) and rule applications for handling logical retraction take constant time. We now consider how often rules can be tried and applied in the worst case.

- rule `already_seen`: The rules for `killl` produce at most kn `killj` constraints (see below). Thus the rule is tried for at most kn times. Duplicate `killj` constraints will be immediately removed. At most $c+n$ different `killj` constraints remain where the rule was not applicable. Note this number is bounded by kn, the number of `killj` constraints produced in the first place.
- rule `go_to_initial`: There are at most kn rule application attempts, one for each `killj` constraint. Since all `killj` constraints are different, there are at most n rule applications, one for each `union` constraint. They produce at most n `killl` constraints.
- rules `killl`: Each of the up to n `killl` constraints involves at most k justifications and therefore the rules produce at most kn `killj` constraints which are subjected to rule `already_seen`.
- rule `killit`: The single `killone` constraint form the query chooses one `killj` constraint with an initial justification, binds this justification and adds a single `dorev` constraint in constant time.
- rule `go_forward`: There are at most n unions that can be tried. At most n justifications can be bound by resulting rule applications.
- rule `remove`: There are at most kn program constraints that can be tried and possibly be removed.
- rule `revive`: There is a single `dorev` constraint. There are at most kn `rem` constraints that can be tried and possibly be replaced by program constraints.
- rule `clean_up`: It applies in constant time for the one `dorev` constraint.

For each rule, this gives at most kn rule application attempts and at most kn rule applications that each need constant time. □

The time complexity of a logical retraction with query $A^{\mathcal{J}} \wedge \texttt{killj(F)} \wedge \texttt{killone}$ in the transformed program $\mathcal{P}^{\mathcal{J}}$ without the cost for partial recomputation of revived constraints is not worse than that the time complexity of the query A without logical retraction in the original program $\mathcal{P}$. It is at least $O(n)$.

We consider the space complexity that is involved in these rule applications.

Lemma 7 (Space Complexity of Overhead for Logical Retraction). Execution of a logical retraction with the query $A^{\mathcal{J}} \wedge \texttt{killj(F)} \wedge \texttt{killone}$ in the transformed program $\mathcal{P}^{\mathcal{J}}$ has an overhead in space proportional to the derivation length n of the query A in the original program $\mathcal{P}$ in the worst case.

Proof. We already know from the proof of Lemma 6 that there can only be up to kn different `killj` constraints at a time. For these, we need space. Rules that do not produce `killj` constraints do not increase the number of constraints and thus do not increase the space needed:

- rule `already_seen` frees space by removing one constraint.
- rule `go_to_initial` produces a `killl` constraint that is immediately reduced.
- rules `killl` always reduce a `killl` constraint to up to k `killj` constraints.
- rule `killit` removes two constraints and adds two constraints.
- rule `go_forward` removes a constraint and adds a built-in constraint.
- rule `remove` frees space by removing one constraint.
- rule `revive` replaces a `rem` constraints by another constraint.
- rule `clean_up` frees space by removing one constraint.

So we need additional space for up to kn `killj` constraints. □

The space complexity of a logical retraction with the query $A^{\mathcal{J}} \wedge \texttt{killj(F)} \wedge \texttt{killone}$ in the transformed program $\mathcal{P}^{\mathcal{J}}$ without the cost for partial recomputation of revived constraints is not worse than the space complexity of the query $A^{\mathcal{J}}$ in the transformed program $\mathcal{P}^{\mathcal{J}}$. The latter has a complexity not better than $O(n)$.

We now summarize our Lemmata on space and time complexity of the overhead for logical retraction in a Theorem.

Theorem 5 (Space and Time Complexity of Overhead for Logical Retraction with Justifications). Given a program $\mathcal{P}$ and a query A with derivation length n. The time complexity of the query A is at least $O(n)$ since a rule application takes at least constant time.

Execution of a query with justifications $A^{\mathcal{J}}$ in the transformed program $\mathcal{P}^{\mathcal{J}}$ has an overhead in space and time proportional to the derivation length n (Proofs in Lemma 4 and Lemma 5).

Execution of a logical retraction with query $A^{\mathcal{J}} \wedge \texttt{killj(F)} \wedge \texttt{killone}$ in the transformed program $\mathcal{P}^{\mathcal{J}}$ has an overhead in space and time proportional to the derivation length n in the worst case. (Proofs in Lemma 7 and Lemma 6).

□

The introduction of justifications increases the space complexity of the query $A^{\mathcal{J}}$ by $O(n)$, but does not increase the time complexity compared to query A. The space and time complexity of the query $A^{\mathcal{J}} \wedge \texttt{killj(F)} \wedge \texttt{killone}$ without the cost for partial recomputation of revived constraints is not worse than the space and time complexity of the query $A^{\mathcal{J}}$. Thus we pay with space for the introduction of justifications, while the time complexity is unchanged. Logical retraction does not further increase time and space complexity. These complexity results are supported by the experiments and benchmarks reported in the paper [Fru17a].

8 Related Work

The idea of introducing justifications into CHR is not new. The thorough work of Armin Wolf on Adaptive CHR [WGG00] was the first to do so. 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. In our approach, we use a straightforward source-to-source transformation and retract constraints one-by-one instead. Adaptive CHR had a low-level implementation in Java [Wol01], while we give an implementation in CHR itself by a straightforward source-to-source transformation that we prove confluent and correct. Moreover we prove confluence of the rule scheme for logical retraction with the rules of the given program. The application of adaptive CHR considered dynamic constraint satisfaction problems (DCSP) only, in particular for the implementation of search strategies [Wol05], while we apply our approach to arbitrary algorithms in order to make them fully dynamic. Adaptive CHR requires a dynamic unification algorithm using justifications [Wol98]. With this extension, built-in constraints of syntactic equality can be handled in the body of rules.

The issue of search strategies was further investigated by Leslie De Koninck et. al. [DKSD08]. They introduce a flexible search framework in $\text{CHR}^\vee$ (CHR with disjunction) extended with rule and search branch priorities. In their work, justifications are introduced into the semantics of $\text{CHR}^\vee$ to enable dependency-directed backtracking in the form of conflict-directed backjumping.

The work of Jeremy Wazny et. al. [SSW03] introduced informally a particular kind of justifications into CHR for the specific application of type debugging and reasoning in Haskell. Justifications correspond to program locations in the given Haskell program. Unlike other work, the constraints in the body of CHR rules have given justifications to which justifications from the rule applications are added at runtime. The justifications are used to find minimal unsatisfiable subsets of constraints.

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 without giving further details that for clause generation, SMCHR supports justifications for constraints that include syntactic equality constraints between variables. According to one reviewer, such SMT solvers also track justifications in the form of a unique SAT variable for each constraint, and rule application is encoded as a SAT clause.

Our work does not need a new semantics for CHR, nor its extension with disjunction, it rather relies on a source-to-source transformation within the standard semantics. Our work does not have a particular application of justifications in mind, but rather provides the basis for any type of application that requires dynamic algorithms.

9 Conclusions

In this paper, the basic framework for CHR with justifications ($\text{CHR}^\mathcal{J}$) has been established, formally analyzed and implemented. We defined a scheme of two rules that introduces justifications into a subset of CHR as a conservative extension. Justifications enable logical retraction of CHR constraints. If a constraint

is retracted, the computation is adapted and continues as if the constraint never was introduced. We proved confluence and correctness of the two-rule scheme that encodes the logical retraction.

We presented a basic implementation using a straightforward source-to-source transformation (available for use online) and gave two classical examples for dynamic algorithms. We improved this implementation. We showed correctness of the implementations for confluent programs.

In the optimized implementation, the computational overhead of introducing justifications and of performing logical retraction, i.e. the additional time and space needed, is proportional to the derivation length in the original program without justifications. This overhead may increase space complexity, but does not change the worst-case time complexity. We pay with space for the introduction of justifications, while a logical retraction does not further increase complexity.

Our approach currently applies to CHR rules without built-in constraints in the body, since built-in constraint solvers do not support removal of constraints. But built-in constraints can be re-implemented in CHR. For the implementations, our correctness results only apply to confluent programs. But then, a non-confluent program does not guarantee identical outcomes in the first place.

Future work could investigate along three themes: dynamic algorithms, implementation aspects and application domains of CHR with justifications.

- First, we would like to research how logical as well as classical algorithms implemented in CHR behave when they become dynamic.
- Second, we would like to further improve the implementation, analyse and benchmark it. Currently, the entire history of removed constraints is stored. It could suffice to remember only a partial history.
- Third, the rule scheme can be extended to support typical application domains of justifications: explanation of derived constraints (for debugging), detection and repair of inconsistencies (for error diagnosis), and nonmonotonic logical behaviors (e.g. default logic, abduction, defeasible reasoning).

Acknowledgements. We thank Daniel Gall for implementing the online transformation tool for the basic implementation of CHR$^{\mathcal{J}}$. We also thank the anonymous reviewers for their helpful suggestions on how to improve the paper. In particular, they asked for correctness results for the implementations.

References

[Abd97] Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In G. Smolka, editor, *CP '97: Proc. Third Intl. Conf. Principles and Practice of Constraint Programming*, volume 1330 of *LNCS*, pages 252–266. Springer, 1997.

[AFM99] Slim Abdennadher, Thom Frühwirth, and Holger Meuss. Confluence and semantics of constraint simplification rules. *Constraints*, 4(2):133–165, 1999.

[Bet14] Hariolf Betz. *A unified analytical foundation for constraint handling rules.* BoD, 2014.

[BM06] Kenneth N Brown and Ian Miguel. Uncertainty and change, chapter 21. *Handbook of Constraint Programming*, pages 731–760, 2006.

[DKSD08] Leslie De Koninck, Tom Schrijvers, and Bart Demoen. A flexible search framework for chr. In *Constraint Handling Rules — Current Research Topics*, volume LNAI 5388, pages 16–47. Springer, 2008.

[DSGH04] Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The refined operational semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *ICLP '04*, volume 3132 of *LNCS*, pages 90–104. Springer, September 2004.

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

[FR18] Thom Frühwirth and Frank Raiser, editors. *Constraint Handling Rules - Compilation, Execution, and Analysis*. BOD, ISBN 9783746069050, January 2018.

[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.

[Fru17a] Thom Fruehwirth. Implementation of Logical Retraction in Constraint Handling Rules with Justifications. In *21st International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2017)*, September 2017.

[Fru17b] 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.

[Fru18] Thom Fruehwirth. Parallelism, concurrency and distribution in constraint handling rules: A survey. *Theory and Practice of Logic Programming*, 18(5-6):759–805, 2018.

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

[RBF09] Frank Raiser, Hariolf Betz, and Thom Frühwirth. Equivalence of CHR states revisited. In F. Raiser and J. Sneyers, editors, *CHR '09*, pages 33–48. K.U.Leuven, Dept. Comp. Sc., Technical report CW 555, July 2009.

[SD04] Tom Schrijvers and Bart Demoen. The K.U.Leuven CHR system: Implementation and application. In Th. Frühwirth and M. Meister, editors, *CHR '04, Selected Contributions*, pages 8–12, May 2004.

[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.

[SSW03] Peter J Stuckey, Martin Sulzmann, and Jeremy Wazny. Interactive type debugging in haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 72–83. ACM, 2003.

[VW10] Peter Van Weert. Efficient lazy evaluation of rule-based programs. *IEEE Transactions on Knowledge and Data Engineering*, 22(11):1521–1534, November 2010.

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

[Wol98] Armin Wolf. Adaptive solving of equations over rational trees. In *Principles and Practice of Constraint Programming*, pages 475–475, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[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.

[Wol05] Armin Wolf. Intelligent search strategies based on adaptive constraint handling rules. *Theory and Practice of Logic Programming*, 5(4-5):567–594, 2005.

REPLY TO REVIEWERS COMMENTS

We substantially revised and extended the paper, in particular, we added several pages about correctness. We corrected all the typos and all minor problems mentioned by the reviewers. Below are answers to the larger issues raised.

Reviewer A:

- In Conclusions "do not support removal of built-in constraints" we made this more explicit in the abstract, introduction, main section and conclusions.

- In Conclusions "adverse effects due to propagation history" after careful reconsideration we realized that the problem is not with propagation rules, but with non-confluent programs. So this is covered by the three newly added subsections on correctness.

- "proofs in abstract semantics, implementation in refined semantics" In the theoretical part we strengthened Theorem 3 for confluent programs. We proved correctness of the basic and of the optimized implementation for confluent programs in two new subsections.

- "indexing, complexity of constraint retrieval" the indexing uses hash tables and arrays, and those admit the required constant times. There is not need to compare keys. As usual in the literature it is assumed that keys, pointers and indices fit into the word memory size of the underlying computing system. It is beyond the scope of the paper to explain the optimizing CHR compilers that admit this constant-time efficiency, there is a wealth of literature about it, and we added two new references. We also clearly state the complexity assumptions in the implementation section and give references that show that these assumptions hold in actual CHR implementations.

- "Prolog disjunction in body of choice rule" we removed the disjunction, it is not necessary in the scope of this paper and it confused several readers.

- "related work on reversible computations" our work is not concerned with reversing computations it is rather about removing parts of a computation using justifications. We still compute final states, we do not return to previous states. For CHR and reversible computing, see e.g.: Zaki, A., Frühwirth, T. W., Abdennadher, S. (2013). Towards inverse execution of constraint handling rules. Theory and Practice of Logic Programming, 13, 4-5.

- "abstract too long" shortened and updated it.

Reviewer B:

- "timings for improved implementation" it is beyond the scope of this paper to present benchmark results. As said in the conclusions, this is future work (but first results can be found in the companion papers from which this article derived).

Reviewer C:

- we made the motivation for source-to-source transformation clearer following the suggestions of the reviewer.

- we improved the section on related work following the suggestions of the reviewer.

- "consider reformulating the confluence results modulo and invariant" there is no need to use such extended machinery, since the results already hold with standard confluence, there are no tricky invariants and equivalence of states is direct.

- "a better motivating example" beyond minimum and shortest path would be incomprehensible in the context of this paper due to its complexity when several non-trivial rules are involved.

- "theoretical results should be extended to the optimized version as well" In the theoretical part we strengthened Theorem 3 for confluent programs. We proved correctness of the basic and of the optimized implementation for confluent programs in two new subsections.

- "interesting applications, realistic use cases" as said in the conclusions, this is future work.