Combining Forward and Backward Propagation

Amira Zaki[1,2], Slim Abdennadher[1], and Thom Frühwirth[2]

[1] German University in Cairo, Egypt {amira.zaki, slim.abdennadher}@guc.edu.eg
[2] Ulm University, Germany thom.fruehwirth@uni-ulm.de

Abstract. Constraint Handling Rules (CHR) is a general-purpose rule-based programming language. This paper studies the forward and backward propagation of rules, and explores the combination of both execution strategies. Forward propagation transforms input to output, while backward propagation uncovers input from output. This work includes a source-to-source transformation capable of implementing a backward propagation of the rules. Furthermore with the addition of annotating trigger constraints, CHR programs can be executed in a strictly-forward, strictly-backward or combined interleaved quasi-simultaneous manner. A programmer should only write one program and then the annotated transformation empowers the multiple execution strategies. The proposed work is useful for automatic implementation of bidirectional search for any search space through the combined execution strategies. Moreover, it is advantageous for reversible bijective algorithms (such as lossless compression/decompression), requiring only one algorithm direction to be implemented.

Keywords: Forward/Backward, Constraint Handling Rules, Bidirectional Search, Combined Propagation, Source-to-source transformation

1 Introduction

A program P can be defined as a series of transitions transforming an input state to an output state, while an inverse (or backward) program P^{-1} is one that uncovers the input given the output [11]. The transition rules transforming input to output are known as forward rules, whereas those reversing output to input are known as backward rules. For example, compression can be considered as a forward program, and decompression is its backward program.

The study of transition directions for the same program captures some interesting program pairs, such as encryption/decryption, compression/decompression, invertible arithmetic functions and roll-back transactions. Despite the relation between inverse computations, it is common practice to maintain two separate programs to perform each transition direction. The two programs are similar and essentially related to one another, however two disjoint programs have to be written. Furthermore, maintaining the relationship between the programs can be a source of errors and inconsistency, since any changes must be reflected in both programs.

To avoid duplication of effort, a user ideally wants to write and maintain a single program. This work facilitates a combination of forward and backward propagation directions automatically for any given program. Programs are written using Constraint Handling Rules (CHR), which is a high-level programming language based on guarded rewrite rules [7]. The language was originally designed to write constraint solvers, however it is a strong and elegant general-purpose language with a large spectrum of applications. Source-to-source transformations extend CHR programs to ones with additional machinery [8]. This work presents a source-to-source transformation, to generate a combined program which is more expressive and complex, featuring two-way execution.

Programming languages themselves vary in the inference direction used; backward chaining languages are highly non-deterministic compared to committed-choice forward chaining languages. Languages supporting deterministic forward and backward computation are known as reversible languages such as the imperative language Janus [16]. Recently, [10] proposed a class of programming languages that generalizes both Constraint Logic and Concurrent Constraint Programming to combine forward and backward chaining, however the work lacks a proper implementation. Prolog uses backward chaining, and can be used to implement two-way programs whose arguments maybe used for input or output. However, implementing such programs is quite tricky, and special care must be taken while implementing with two-way predicates and operators.

For example, the run-length encoding algorithm is a simple data compression technique, where consecutive runs of characters within a text are packed and stored as a single character followed by its count. The text '`aaaaabccc`' (wrapped in a `compress/1` constraint) is compressed into '`a5b1c3`' (in a `result/1` constraint). The run-length encoding algorithm can be expressed in CHR as shown below, where `pack/2` is used to pack consecutive letter runs and `comp/3` to accumulate the encoding. This program encapsulates the combined forward and backward nature of the algorithm, and two-way execution is explored through the transformations described this work.

```
start @ compress(In) <=> comp(In,[],[]).
run-end @ comp([H1,H2|T],Run,Acc) <=> H1\=H2 | Run2=[H1|Run],
    pack(Run2,PackRun), append(Acc,[PackRun],Acc2), comp([H2|T],[],Acc2).
run-cont @ comp([H1,H2|T],Run,Acc) <=> H1=H2
    | Run2=[H1|Run],comp([H2|T],Run2,Acc).
last-char @ comp([H],Run,Acc) <=> Run2=[H|Run],
    pack(Run2,PackRun), append(Acc,[PackRun],Acc2), result(Acc2).
end @ comp([],_,_) <=> result([]).
```

This work presents a source-to-source transformation that captures the backwards operational semantics of CHR for range-restricted programs. The transformation was introduced earlier [17] however in this work it is revisited and expressed more formally. The previously mentioned run-length encoding program is transformed into a combined form as presented in this work. Then for a string '`aaaaabccc`', the program can be forwardly run by: '`fwd, compress([a,a,a,a,a,b, c,c,c])`' to produce the compressed form: '`result([[a,5], [b,1],[c,3]])`'. Similarly, a backward run can be attained through a query: '`bck, result([[a,5], [b,1],[c,3]])`' to decompress the sub-lists to: '`compress([a,a,a,a,a,b,c,c,c])`'.

Furthermore this work introduces the addition of annotating trigger constraints, to empower the execution of CHR programs in a strictly-forward, strictly-backward or interleaved quasi-simultaneous bidirectional manner. This means that only one program is written and the annotated transformation enables multiple execution schemes. This is useful for solving bijective algorithms and the aforementioned inverse computation pairs. The paper extends by showing how the combined programs can be extended to facilitate bidirectional search.

Previous work, such as [1], tend to use CHR as a language for abduction. However, the problem is that the user has to write two different programs for deductive and abductive reasoning. This work presents a technique to write a program once, then facilitate different reasoning directions.

The paper proceeds by recalling background information about CHR in Section 2. In Section 3, the combined two-way programs are presented through source-to-source transformations, and then the extension for bidirectional search is given in Section 4. This is followed by an application in Section 5. The paper concludes by some remarks and future work in Section 6.

2 Constraint Handling Rules

2.1 Syntax

Constraint Handling Rules (CHR) [7,9] is a high level, committed choice, rule-based programming language. It consists of guarded rewrite rules that perform conditional transformation of a multi-set of constraints. It distinguishes between two types of constraints; built-in constraints which are predefined by the host language and other user-defined CHR constraints which are declared as functor/arity pairs. A generalized CHR simpagation rule is given as:

$$\textit{rule-id} \ \texttt{@} \ H_k \setminus H_r \ \Leftrightarrow \ G \ | B$$

Every rule has an optional unique identifier preceding it given by *rule-id*. H_k and H_r are a conjunction of one or more CHR constraints; known as the kept and removed head constraints respectively. The guard G is an optional conjunction of built-in constraints. The body of the rule B consists of a conjunction of both built-in and CHR constraints.

Simplification and propagation rules are two other rule types which are special cases of simpagation rules. Simplification rules have no kept head constraints, and propagation rules have no removed head constraints. They are of the forms:

$$simplification\text{-}id \ \texttt{@} \ H_r \ \Leftrightarrow \ G \ | \ B$$

$$propagation\text{-}id \ \texttt{@} \ H_k \ \Rightarrow \ G \ | \ B$$

Constraint Handling Rules with Disjunction (CHR$^\vee$) [2] is an extension of CHR featuring disjunctive rule bodies to allow a backtrack search over alternatives.

The rules are similar to the rule forms described above, however the rule body can be composed of two or more disjunctive bodies ($B_1 \vee B_2$). For example a CHR$^\vee$ simpagation rule is of the form:

$$disjuntive\text{-}id \; \texttt{@} \; H_k \setminus H_r \;\; \Leftrightarrow \;\; G \;\; \texttt{|} \;\; B_1 \;\; \texttt{;} \;\; B_2$$

2.2 Operational Forward Semantics

The behavior of a CHR program is modeled through an operational semantics, in terms of a state transition system. The very abstract semantics (ω_{va}) is a state transition system, where a transition corresponds to a rule application and states represent goals consisting of a conjunction of CHR and built-in constraints. An initial state is an arbitrary one and a final state is a terminal one where no further transitions are possible. The ω_{va} semantics includes one rule which is shown below, where P is a CHR program and $\mathcal{CT}$ is the constraint theory for the built-in constraints. The body of a rule (B) and C consist of both built-in and CHR constraints, moreover H_k and H_r are a conjunction of CHR constraints, while G is a conjunction of built-in constraints.

Apply

$$(H_k \wedge H_r \wedge C) \mapsto^{r}_{apply} (H_k \wedge G \wedge B \wedge C)$$

if there is an instance of a rule r in P with new local variables $\bar{x}$ such that:

$$r \; @ \; H_k \setminus H_r \Leftrightarrow G \mid B \text{ and } \mathcal{CT} \models \forall(C \rightarrow \exists \bar{x} G)$$

The extended transition system for CHR$^\vee$ operates on a disjunction of CHR states known as a configuration: $S_1 \vee S_2 \vee \cdots \vee S_n$. The original apply transition is applicable to a single state. An additional split transition is applicable to any configuration containing a disjunction. It leads to a branching derivation entailing two states, where each state can be processed independently.

Split

$$((H_1 \vee H_2) \wedge C) \vee S \mapsto_{\vee} (H_1 \wedge C) \vee (H_2 \wedge C) \vee S$$

However, these semantics are highly non-deterministic and thus more refined semantics are needed for the actual implementation of CHR compilers[5]. The order of constraint execution and rule application determine how a derivation proceeds and is defined by the implemented operational semantics. Starting with the same initial query, multiple derivations are possible. If all derivations ultimately lead to the same goal, then the program is known as a confluent one.

2.3 Operational Backwards Semantics

The forward ω_{va} semantics models a forward rule application on an initial state to a final state. The inverse of this rule application is defined by a *backwards* semantics ω_b that reverses a final state to an initial state. This semantics was formally introduced in [17], and it is typically the same as the apply transition but with exchanging the left and right hand side states of the transition.

Backwards

$$(H_k \land G \land B \land C) \mapsto^r_{back} (H_k \land H_r \land C)$$

if there is an instance of a rule r in P with new local variables $\bar{x}$ such that:

$$r \ @ \ H_k \setminus H_r \Leftrightarrow G \mid B \text{ and } \mathcal{CT} \models \forall(C \rightarrow \exists\bar{x}G)$$

The semantics works by undoing each step of the forward execution. However without any external knowledge on how to proceed with the inverse tree, the backwards semantics only shows that any original state can be uncovered. In fact, inverse programs are normally non-confluent ones.

3 Combined CHR Programs

This work builds upon the K.U. Leuven system [9], which uses CHR with Prolog as the host language. To introduce the two-way execution, the contribution of the existing Leuven system is presented and source-to-source transformations are given to augment additional machinery on-top of the existent CHR system.

3.1 General Formulation

The first part of this work is to define a transformation of a CHR program for forward/backward execution. Programs are intended to be written once and then executed in several ways; data supplied is considered as input or output depending on the program direction used. Figure 1 represents the relation between input and output and the expected transitions of the two-way program.

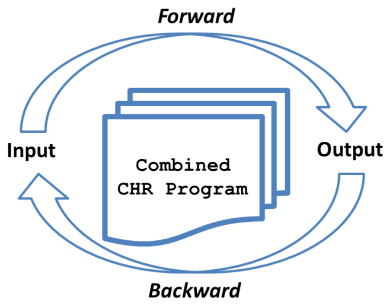

Fig. 1. Transitions between input and output

The cardinality of the program that transforms the input to output, decides on the properties of the two-way program required. If the relation is one-to-one, then for every output there exists only one input and vice versa. This makes the backward transition quite straight-forward, since for every output there is only one possible input that caused it. These relations would require a direct forward/backward execution mechanism. This is especially useful for bijective functions such as loss-less compression/decompression and encryption/decryption.

On the other hand, a list sorting program transforms several permutations into the same sorted output. Hence the forward transition has a many-to-one cardinality, and therefore its backward transition (shuffling a sorted list) is one-to-many. Due to the committed-choice nature of CHR and the deterministic implementation of the Leuven system, it would never reach all transition possibilities. Thus, it requires using a source-to-source transformation [6] that fully explores the search-space to reach all final states.

Therefore, for every transition direction two execution strategies are required; a direct one-way execution and an exhaustive execution. Annotations are added to the combined program, to decide on the chosen execution strategy. The annotation involves adding a kept head constraint to each program rule, where its presence will denote the activation of this rule. The annotation will involve four CHR constraints: `fwd/0`, `fwd-ex/0`, `bck/0` and `bck-ex/0`. The following subsections highlight the necessary changes that are needed to transform a CHR program into a combined two-way one with control terms that steer the direction. The summary of the transformations can be depicted in Figure 2.

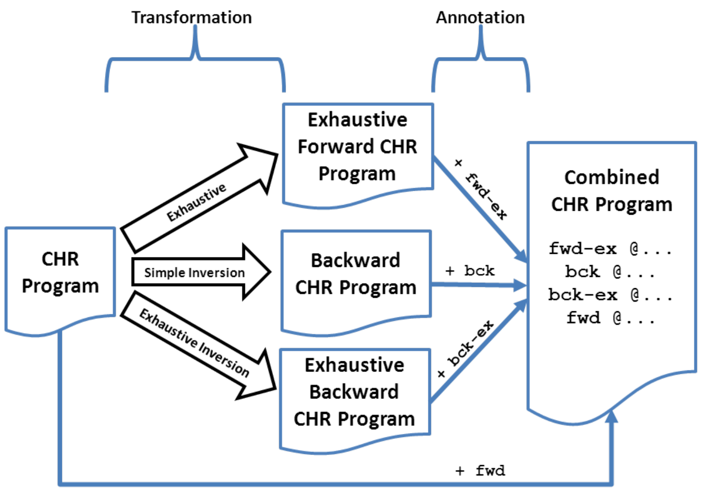

Fig. 2. Bidirectional CHR Transformations

3.2 Forward CHR

In order to change the program execution flow, source-to-source transformations are used to facilitate a straight-forward implementation on top of existing CHR implementations whilst exploiting the optimizations of current CHR compilers [15]. The normal execution of a committed-choice CHR program can be transformed into one featuring exhaustive completion to fully explore a goal's search space [6].

Directly Forwards The CHR Leuven system follows a refined operational semantics with additional fixed orders for explored goals and chosen program rules. It applies the program rules on a goal, until a fixed point is reached. Thus this provides the direct forwards execution of the combined program.

A constraint `fwd/0` is introduced as a kept head constraint to every program rule. Thus for every generalized CHR simpagation rule, a corresponding annotated rule is added of the form shown below. For other rules, the missing constraints are non-existent accordingly (i.e H_k or H_r).

$$\textit{fwd-simpagation} \; \texttt{@} \; \texttt{fwd} \; , \; H_k \; \backslash \; H_r \; \Leftrightarrow \; G \; | \; B$$

Example 1 Run-length Encoding *The compression algorithm presented in the introduction can be rewritten using forward annotation as shown below. For a string* ‘`aaaaabccc`’*, the program is forwardly run by:* ‘`fwd,compress([a,a,a,a,a,b,c,c,c])`’ *to yield the compressed form:* ‘`result([[a,5], [b,1],[c,3]])`’*.*

```
start @ fwd \ compress(In) <=> comp(In,[],[]).
run-end @ fwd \ comp([H1,H2|T],Run,Acc) <=> H1\=H2 | Run2=[H1|Run],
     pack(Run2,PackRun), append(Acc,[PackRun],Acc2), comp([H2|T],[],Acc2).
run-cont @ fwd \ comp([H1,H2|T],Run,Acc) <=> H1=H2
     | Run2=[H1|Run],comp([H2|T],Run2,Acc).
last-char @ fwd \ comp([H],Run,Acc) <=> Run2=[H|Run], pack(Run2,PackRun),
     append(Acc,[PackRun],Acc2), result(Acc2).
end @ fwd \ comp([],_,_) <=> result([]).
```

Exhaustive Forward For non-confluent programs, overlapping sets of rule constraints and the order of constraints within rules and queries entail different derivation paths to several possible outputs. The exhaustive transformation [6] was proposed to allow full space exploration for any CHR program to reach all possible solutions to a query. It changes a CHR derivation into a search tree with disjunctive branches to reach all leaves.

A `depth/1` constraint is added to represent the current depth of the search tree. The transformation annotates constraint occurrences within the rules with two additional arguments; one to denote the occurrence number and the other to represent the current depth within the search tree. Details of the transformation for forward execution will not be revisited here due to space limitations, and can be directly referred to in [6]. However it will be presented for backward execution in the next section, since it is a modification of [6]. Thus for a forward CHR program, the transformation is applied and the resulting CHR rules are annotated with a `fwd-ex/0` kept head constraint.

Example 2 Sets of Cards *Given N cards, each represented with a* `card/1` *constraint, a simple program can be written to select three cards whose sum equals* 12*, to form a set (*`set/3`*) using the predefined* `sumlist/2` *list predicate.*

```
select @ card(A), card(B), card(C) <=> sumlist([A,B,C],12) | set(A,B,C).
```

Running the program with a query '`card(1),card(2),card(3),card(4),card(5), card(6),card(7),card(8),card(9),card(10)`', will result in: '`card(10), card(8), card(7),card(6),set(5,4,3),set(9,2,1)`'. Due to the implementation of the CHR compiler, only one result is reached. However there are multiple other sets that can be assembled from those 10 cards. The forward program is transformed according to the exhaustive transformation to produce the below program (where `card(X,_,_)` is equivalent to `card(X)`):

```
src-mod @ fwd-ex \ depth(Z), card(A,1,Z), card(B,2,Z), card(C,3,Z)
    <=> sumlist([A,B,C],12) | set(A,B,C).
assign @ fwd-ex, depth(Z) \ card(A)
    <=> card(A,0,Z); card(A,1,Z); card(A,2,Z); card(A,3,Z).
rest @ fwd-ex, depth(Z) \ card(A,0,Z1) <=> Z1 < Z | card(A).
pruning @ fwd-ex \ end, card(A,_,_), card(B,_,_), card(C,_,_)
    <=> sumlist([A,B,C],12) | fail.
```

The exhaustive forward program can be run with the same query as before but adding the appropriate `fwd-ex` trigger. The query's execution gets transformed into a derivation tree, producing all possible card set combinations.

3.3 Backward CHR

Directly Backwards The backwards semantics ω_b can be achieved through a source-to-source transformation of the CHR program. The transformation idea was introduced in [17] but will be formalized in this paper.

Definition 1. Backwards Transformation *Every range restricted rule of the form (r @ $H_k \setminus H_r \Leftrightarrow G \mid B$) in program P (where $B = B_b \wedge B_c$ representing the built-in and CHR constraints respectively), an inverse rule* `in-r` *is added to the transformed program P^{-1} of the form (where* `bck` *is an annotating trigger constraint):*

$$\textit{in-r @ bck, } H_k \setminus B_c \;\Leftrightarrow\; B_b,\; G \mid H_r$$

Applying the backwards transformation on the cards example, would require one backward rule as shown below:

```
bck-select @ bck \ set(A,B,C)<=>sumlist([A,B,C],12)|card(A),card(B),card(C).
```

Similarly, the previous run-length encoding program (Example 1) can be transformed using the backwards transformation as follows:

```
bck-start @ bck \ comp(In,[],[]) <=> compress(In).
bck-run-end @ bck \ comp([H2|T],[],Acc2) <=>
    Run2=[H1|Run], pack(Run2,PackRun), append(Acc,[PackRun],Acc2), H1\=H2
    | comp([H1,H2|T],Run,Acc).
bck-run-cont @ bck \ comp([H2|T],Run2,Acc) <=> Run2=[H1|Run], H1=H2
    | comp([H1,H2|T],Run,Acc).
bck-last-chr @ bck \ result(Acc2) <=>
    Run2=[H|Run],  pack(Run2,PackRun), append(Acc,[PackRun],Acc2)
    | comp([H],Run,Acc).
bck-end @ bck \ result([]) <=> comp([],_,_).
```

Decompression of the encoded message can be easily attained by a backwards transition from output to input. Thus a query '`bck, result([[a,5], [b,1],[c,3]])`' decompress the sub-lists to: '`compress([a,a,a,a,a,b,c,c,c])`'.

Exhaustive Backward The completeness of the backwards transformation relies on the high-level non-determinism of the ω_{va} semantics. The completion fails when implementing on top of current CHR systems. Thus for implementation, the backwards transformation is coupled with an exhaustive execution transformation [6]. To illustrate why this is necessary consider the next sorting example.

Example 3 *(Exchange sort) In CHR, constraints of the form* `n(Index,Value)` *can be sorted by exchanging any pair of constraints with an incorrect order. This is possible through a forward program consisting of a single simplification rule:*

```
sort @ fwd \ n(I,V),n(J,W) <=> I>J,V<W | n(I,W),n(J,V).
```

Using the defined transformation, the program becomes:

```
in-sort @ bck \ n(I,W),n(J,V) <=> I>J,V<W | n(I,V),n(J,W).
```

The two-way program sorts a query '`fwd,n(0,9),n(1,1), n(2,5)`' to ordered numbers represented as '`n(0,1),n(1,5), n(2,9)`'. On the other hand, a query '`bck,n(0,1),n(1,5),n(2,6)` uncovers the permutation '`fwd,n(0,9),n(1,4),n(2,1)`'. This is a correct input, but not necessarily the exact one used in the forward run. The reason is that sorting is a many-to-one function, where permutations of unsorted lists derive the same sorted list. The inverse of sorting problem is a shuffle operation which generates all possible permutations of the ordered list. This cannot be achieved here as the backwards transition generates only one permutation.

The transformation required to generate exhaustive backward program rules is shown next. All the generated rules are annotated with a `bck-ex` constraint to distinguish them within the two-way program. All unannotated inverse rules (`in-r @` $H_k \setminus B_c \Leftrightarrow B_b,\ G \mid H_r$) in program P^{-1} are transformed as described by the upcoming Definition 2.

Definition 2. Exhaustive Backwards Transformation *A transformed inverse exhaustive program P^{-T} is defined for a program P by the three following steps (adapted from [6] but with no pruning of intermediate states).*

1. *Each constraint $c(X_1, ..., X_n)$ in a forward program's B_c constraints is changed to $c^t(X_1, ..., X_n, y, Z)$, such that constraint occurrences within the program (where m is the total number of occurrences) are annotated with an argument y and depth Z. y represents the yth occurrence of the constraint c, i.e. $y \in [1, m]$. Thus, for every constraint $c(X_1, ..., X_n)$ that appears in the forward program's rule body, an assignment rule is added to the transformed program, defined as follows:*
   `assign`$_c$ `@ bck-ex,` $depth(Z) \setminus c(X_1, ..., X_n)$
   $\Leftrightarrow\ c^t(X_1, ..., X_n, 0, Z) \vee ... \vee c^t(X_1, ..., X_n, m, Z)$
2. *For every rule ($H_r \Leftrightarrow G \mid B_b, B_c$) in a forward program, with $B_c = c_1(X_{11}, ..., X_{1n_1}), ..., c_l(X_{l1}, ..., X_{ln_l})$, a modified source rule is added to the transformed program, as follows:*
   `in-r`$_t$ `@ bck-ex` $\setminus depth(Z), {c_1}^t(X_{11}, ..., X_{1n_1}, y_1, Z), ...,$
   ${c_l}^t(X_{l1}, ..., X_{ln_l}, y_l, Z)\ \Leftrightarrow\ B_b, G \mid H_r, depth(Z + 1)$

3. *An additional rule is needed to reset unmatched constraints if a newly state in the tree is derived. Hence, for every constraint $c(X_1, ..., X_n)$ that appears in B_c, a reset rule is added to the transformed program:*
   $\texttt{reset}_c$ `@ bck-ex,` $depth(Z) \setminus c^t(X_1, ..., X_n, 0, Z') \Leftrightarrow Z' < Z \mid c(X_1, ..., X_n)$

Example 4 *(Exchange sort - Revisited) Applying the newly defined transformation on the exchange sort of Example 3, will generate the following rules:*

```
assign-a @ bck-ex, depth(Z) \ n(X,Y)
    <=> n_t(X,Y,0,Z); n_t(X,Y,1,Z); n_t(X,Y,2,Z).
in-sort-t @ bck-ex \ depth(Z),n_t(I,W,1,Z),n_t(J,V,2,Z)
    <=> I>J, V<W | n(I,V), n(J,W), depth(Z+1).
reset-a @ bck-ex, depth(Z) \ n_t(X,Y,0,Z1) <=> Z1 < Z | n(X,Y).
```

The transformed rules can be run with the sorted input: `bck-ex, depth(0), n(0,1), n(1,5),n(2,9)`. It generates several results, which form the complete set of all permutations of those three numbers. However there exists several redundancies; the intensive use of disjunction produces several duplicate states which are revisited multiple times. The backward run is terminating, and the use of a breadth-first strategy covers the entire search space. The reason for this is that the number of permutations of a list is finite.

4 Interleaved Forward/Backward Propagation

The combined two-way program enables either a strictly forward or strictly backward execution depending on the used trigger. However, we further propose an additional transformation towards a combined interleaved execution, which is inspired from bidirectional search. Bidirectional search tries to find the shortest path to a node/element by running two simultaneous searches. It involves one forward search from the initial state, and one backward search from the goal state. The search stops when the two searches reach the same state, somewhere in the middle. In many problems, bidirectional search can dramatically reduce the amount of required exploration [14]. The two-way CHR programs can be modified to implement a bidirectional search for a goal. Instead of running a transition in a strictly forward or strictly backward manner, we introduce a technique to have an interleaved forward and backward manner to achieve a combined quasi-simultaneous two-way execution.

For clarity, bidirectional search is exemplified with direct forwards and backwards transitions. The technique can also be applied to the exhaustive variants, but it makes the presentation too long for the scope of this paper.

Example 5 List Searching *Determining whether an element is found within a list can be performed in CHR as shown below. A constraint* `find/2` *is used to search in the first argument (a list) for the second argument and a constraint* `found/1` *denotes that it has been found. A query* `fwd, find([0,1,2,3,4],3)` *would reach the goal* `found(3)`.

```
end @ fwd \ find([X],Y) <=> X=Y, found(Y).
middle @ fwd \ find([X|_],Y) <=> X=Y | found(Y).
search @ fwd \ find([X|Xs],Y) <=> X\==Y | find(Xs,Y).
```

The backward search from a found element, constructs arbitrary lists containing this element. The direct backward rules are given as:

```
in-end @ bck \ found(Y) <=> X=Y | find([X],Y).
in-middle @ bck \ found(Y) <=> X=Y | find([X|_],Y).
in-search @ bck \ find(Xs,Y) <=> X\==Y | find([X|Xs],Y).
```

Due to the chosen direct (non-exhaustive) execution, the second rule (`in-middle`) becomes unreachable in this context and these rules form a non-confluent program; the first two rules have the exact same rule heads and guards. One way to resolve this problem is to use the previously introduced exhaustive execution. Alternatively, since these rules are single-headed with the same guards, then it is sufficient to use Clark's completion and merge the two rules into one. For clarity and to save writing space, the second representation is preferred here over the exhaustive execution. Thus the rules `in-end` and `in-middle` are equivalent to:

```
in-end-middle @ bck \ found(Y) <=> X=Y | find([X],Y) ; find([X|_],Y).
```

Due to the lossy nature of the program the other un-found values are lost. Thus a query `bck,found(3)` would reach several lists with unknown filler values. Some of the backward goals reached are: `find([3],3)`, `find([3,_],3)`, `find([_,3,_],3)`, etc.

For the automatic implementation of a bidirectional search, the idea is to change the execution of these rules such that it follows alternating forward and backward transitions.

Definition 3. Bidirectional Transform *A combined two-way program P^{-T} can be transformed to enable quasi-simultaneous bidirectional search by the following steps:*

1. *Trigger constraints* `bck` *and* `fwd` *should not be kept head constraints. They must be consumed by the rules, and on rule application, the opposite direction trigger is added. Thus forward rules (*`fwd` , $H_k \setminus H_r \Leftrightarrow G \mid B$*) should be changed into:*

$$H_k \setminus \texttt{fwd},\ H_r \Leftrightarrow G \mid B, \texttt{bck}$$

Similarly, backward rules (`bck,` $H_k \setminus B_c \Leftrightarrow B_b,\ G \mid H_r$*) become:*

$$H_k \setminus \texttt{bck},\ B_c \Leftrightarrow B_b,\ G \mid H_r, \texttt{fwd}$$

2. *Constraints of the backward rules must be differentiated from the forward rules, such that each search direction operates on different goals until they meet. Thus every constraint $c(X_1, ..., X_n)$ in the backward rules is changed to $c^b(X_1, ..., X_n)$.*
3. *A unification rule must be added to halt the execution once both search goals can be unified with one another. Thus given a forward goal $c(X_1, ..., X_n)$ and a backward goal $c^b(Y_1, ..., Y_n)$, a possible unifying rule would be of the form:*

$$\texttt{unify @ } c(X_1, ..., X_n),\ c^b(Y_1, ..., Y_n) \Leftrightarrow unifiable(c(X_1, ..., X_n), c^b(Y_1, ..., Y_n), \_) \\ \mid \texttt{write('Bidirectionally found!')}.$$

Therefore the interleaved quasi-simultaneous bidirectional list search program becomes as shown below; all constraints of backward rules are distinguished with (`_b`).

```
end @ fwd, find([X],Y) <=> X=Y, found(Y), bck.
middle @ fwd, find([X|_],Y) <=> X=Y | found(Y), bck.
search @ fwd, find([X|Xs],Y) <=> X\==Y | find(Xs,Y), bck.
unify @ find(X,Y), find_b(Z,Y) <=> unifiable(Z,X,_)
    | write('Bidirectionally found!').
in-end-middle @ bck, found_b(Y) <=> X=Y
    | (find_b([X],Y); find_b([X|_],Y)), fwd.
in-search @ bck, find_b(Xs,Y) <=> X\==Y
    | find_b([X|Xs],Y), fwd.
```

Searching for an element 3 in a list $[0, 1, 2, 3, 4]$ can be performed by the bidirectional search program, with a query `find([0,1,2,3,4],3), found_b(3), fwd`. The derivation for this query would be as shown below, while underlining the matched constraints (a trace is also shown in Figure 3): $\underline{\texttt{fwd, find([0,1,2,3,4],3)}}$, `found_b(3)`

$\mapsto_{\texttt{search}}$ `find([1,2,3,4],3),` $\underline{\texttt{bck, found\_b(3)}}$
$\mapsto_{\texttt{in-middle}}$ $\underline{\texttt{find([1,2,3,4],3), fwd,}}$ `find_b([3,_],3)`
$\mapsto_{\texttt{search}}$ `find([2,3,4],3),` $\underline{\texttt{bck, find\_b([3,\_],3)}}$
$\mapsto_{\texttt{in-search}}$ $\underline{\texttt{find([2,3,4],3)}}$`, fwd,` $\underline{\texttt{find\_b([\_,3,\_],3)}}$
$\mapsto_{\texttt{unify}}$ `write('Bidirectionally found!').`

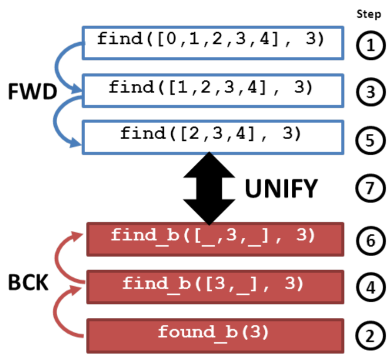

Fig. 3. Bidirectional search trace: '`fwd, find([0,1,2,3,4],3), found_b(3)`'

5 Application for Combined Programs

Another application of the proposed work is for reasoning. Reasoning is the process of using existing knowledge to infer conclusions, speculate predictions, and create explanations. The philosopher C. S. Pierce distinguished between three kinds of reasoning; deduction, induction and abduction [13]. Deduction involves applying rules to specific cases to deduce a certain result, while induction is reasoning which infers a rule from a case and result. Abduction is a kind of backward reasoning which infers a case from the rule and result.

The relation between abduction and reverse deduction has been studied in several works to highlight the difference between them [12]. However, it has also been argued that abduction is a form of reversed deduction and that there is a duality in the explanation of abduction and deduction [3].

The combined two-way programs capture the duality relation between deduction and abduction, and produce a powerful reasoning program. The reasoner exploits existent knowledge to infer conclusions and speculate predictions for observed phenomena. Logic theories that describe the real world are modeled in CHR. Then the exhaustive forward transformation facilitates exhaustive exploration of a query's search space and thus enables deductive reasoning. Furthermore, an exhaustive backward execution of the modeled CHR programs empowers abductive reasoning.

It is not the first time that CHR has been used for abductive reasoning. In [1], logic programs containing Horn clauses are expressed in CHR while differentiating between intensional predicates and extensional ones to perform abductive reasoning. However in this work, all logic clauses including non-Horn ones can be modeled in CHR. Moreover the framework empowers both abductive and deductive reasoning. This is not possible with [1] since the used representation relied heavily on the underlying meaning of abduction and manually gathered similar rule bodies as disjunctive rule bodies.

5.1 Modeling

In order to use the annotated transformations for two-way reasoning, the modeling of logic theories in CHR must first be formalized. A logic theory T is a set of well-formed formulae, where each formula is an implication of the form $A \to B$, and A and B are conjunctions of one or more literals. Logic implications are mapped in a one-to-one manner to CHR simplification rules. The mapping is quite similar to [1], nonetheless in our model both A and B can be conjunctions.

The model is defined by representing literals with CHR constraints and built-ins. Due to the syntax of CHR, two filtering functions are also defined: a `chr/1` function that extracts the predicates/constraints from a set of literals and a `built/1` function that extracts the built-in expressions from a set of literals.

Thus, every implication of the form $A \to B$ is modeled as a forward CHR simplification rule of the form:

$$\texttt{fwd} \setminus \mathit{chr}(A) \Leftrightarrow \mathit{built}(A) \mid B$$

Given the following logic theory which defines some family relations [1]:

Example 6 $father(F, C) \to parent(F, C)$, $mother(M, C) \to parent(M, C)$, $parent(P, C1), parent(P, C2), C1 \neq C2 \to sibling(C1, C2)$

It is transformed into the following annotated forward CHR rules:

```
fwd \ father(F,C) <=> parent(F,C).
fwd \ mother(M,C) <=> parent(M,C).
fwd \ parent(P,C1), parent(P,C2) <=> C1\=C2 | sibling(C1,C2).
```

Integrity constraints can be added to the modeled program to provide semantic optimization to the reasoner. Since these rules ensure the integrity, they are not involved in any of the transformations and thus should not be annotated with any trigger constraints. For Example 6, the following integrity constraints can be added:

```
father(F1,X) \ father(F2,X) <=> F1=F2.
mother(M1,X) \ mother(M2,X) <=> M1=M2.
person(P,G1)\ person(P,G2) <=> G1=G2.
father(F,X) ==> person(F,male), person(X,_).
mother(M,X) ==> person(M,female), person(X,_).
```

An extensional (trigger-less) introduction rule is required to add all the facts into the constraint store, to be introduced with a `start` constraint in any query:

```
start ==> parent(john,mary), father(john,peter), mother(jane,mary),
          person(john,male), person(mary,female), person(paul,male),
          person(peter,male), person(jane,female).
```

To ensure a closed world, the set of hypothesis facts for a given predicate need to be pruned. Closing rules (also without trigger constraints) are added for these predicates [1]. For a predicate p/n defined by $p(t_1^1,\ldots,t_n^1),\ldots,p(t_1^k,\ldots,t_n^k)$, a closing rule is required as a propagation rule shown below:

$$p(x_1,\ldots,x_n) \Rightarrow (x_1 = t_1^1, \wedge, \ldots, \wedge x_n = t_n^1) \quad \vee \cdots \vee (x_1 = t_1^k, \wedge, \ldots, \wedge x_n = t_n^k)$$

To restrict the $person/2$ predicate of Example 6, a closing rule would be added as shown below:

```
person(X,Y) ==> (X=john, Y=male);(X=peter, Y=male); (X=paul, Y=male);
                (X=jane, Y=female);(X=mary, Y=female).
```

5.2 Strictly Forward

Due to the modeling of non-Horn clauses, the normal execution of CHR would not yield deductive reasoning. However, transforming the program to an exhaustive variant would ensure the completeness of the search-space. Using the transformation, it is possible to start from an initial query and deduce all possible derivations to goals.

Thus for deductive reasoning, only rules representing the main transformed implications (i.e. those annotated with `fwd`) are transformed into rules featuring exhaustive execution using the exhaustive transformation. These other rules maintain certain properties for the modeling, hence they need not be transformed.

The three implication rules of the family example can be modeled into CHR and then transformed into their exhaustive executing variant with the constraint trigger `fwd-ex`. Using the initial knowledge that John is the father of Peter and Mary and that Jane is the mother of Mary, one can deduce that Mary and Peter are siblings and that Paul, Jane, Peter, Mary and John are all persons. This deduction can be reached using a query '`fwd-ex, start, father(john,peter), father(john,mary), mother(jane,mary),depth(0)`', to produce the final state:

```
sibling(mary,peter), person(paul,male), person(jane,female),
person(peter,male),  person(mary,female), person(john,male).
```

5.3 Strictly Backward

For abductive reasoning, the exhaustive backwards transformation is performed for, again, only the main annotated CHR rules representing transformed implications from the logic theory.

Abductive reasoning involves deriving hypotheses about certain predicates that are incompletely defined; these are known as abducible predicates. Thus to include the notion of abducibles in the proposed model, only the closing rules of non-abducible predicates are retained (as forward and untransformed rules); other closing rules are completely removed from the program. All other integrity constraint rules and extensional introduction rules are also kept unchanged in the abductive program.

In the family example, predicates `father` and `mother` are abducible but not `person`. Thus the abductive program should contain only one closing rule for `person`. Executing the query '`sibling(paul,mary),bck-ex`' with the abductive program, arrived to the following two possibilities: `father(john,paul)`, or `mother(jane,paul)`.

The goals present two different abductive explanations as to how Paul and Mary are siblings, i.e. either John is the father of Paul or that Jane is the mother of Paul. Furthermore, the abductive query '`sibling(goofy,mary)`' fails because `person` is not abducible. These results match those reached by the abductive CHR modeling of [1].

6 Conclusion

The paper presents a combined perspective for Constraint Handling Rules based on a source-to-source transformation. It involves transforming CHR programs into ones capable of both forward and backward propagation, either in a direct committed-choice manner or in an exhaustive full-space explorative manner. The combination is especially useful for implementing high-level bijective functions, such as encryption/decryption and compression/decompression algorithms, for implementing quasi-simultaneous bidirectional search algorithms and for exploiting dual definitions of reasoning, such as for deduction and abduction.

For future work, an evaluation of the bidirectional search is needed to determine how bidirectionality reduces the amount of required exploration. The search implementations can also be extended to experimenting with different search directions, such as the breadth-first traversal of CHR [4]. Moreover, the proposed reasoning framework is to be compared with other abductive and deductive systems and to evaluate the attained results. Moreover, it could be possible to include the notion of probabilistic abduction by encoding the probabilities in the search tree generated by the exhaustive transformation. Then once the transformation is defined, it would be compared with other implementations of probabilistic abductive logic programs.

References

1. Slim Abdennadher and Henning Christiansen. An experimental CLP platform for integrity constraints and abduction. In *FQAS '00: Proc. 4th Intl. Conf. Flexible Query Answering Systems*, pages 141–152, 2000.
2. Slim Abdennadher and Heribert Schütz. CHR$^{\vee}$: A flexible query language. In *Flexible Query Answering Systems*, volume 1495 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, 1998.
3. Luca Console, Daniele Theseider Dupr, and Pietro Torasso. On the relationship between abduction and deduction. *J. Log. Comput.*, 1(5):661–690, 1991.
4. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. Search strategies in CHR(Prolog). In T. Schrijvers and Th. Frühwirth, editors, *Proceedings of the 3rd Workshop on Constraint Handling Rule*, pages 109–124. K.U.Leuven, Department of Computer Science, Technical report CW 452, 2006.
5. 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, *Proceedings of the 20th International Conference on Logic Programming*, pages 90–104, 2004.
6. Ahmed Elsawy, Amira Zaki, and Slim Abdennadher. Exhaustive execution of chr through source-to-source transformation. In Maurizio Proietti and Hirohisa Seki, editors, *Logic-Based Program Synthesis and Transformation*, volume 8981 of *LNCS*, pages 59–73. 2015.
7. Thom Frühwirth. *Constraint Handling Rules.* Cambridge University Press, 2009.
8. Thom Frühwirth and Christian Holzbaur. Source-to-source transformation for a class of expressive rules. In F. Buccafurri, editor, *Joint Conference on Declarative Programming APPIA-GULP-PRODE 2003 (AGP 2003)*, pages 386–397, 2003.
9. Thom Frühwirth and Frank Raiser, editors. *Constraint Handling Rules: Compilation, Execution, and Analysis.* Books on Demand, March 2011.
10. Rémy Haemmerlé. On combining backward and forward chaining in constraint logic programming. In *Proceedings of 16th International Symposium on Principles and Practice of Declarative Programming (PPDP 2014)*, 2014.
11. Cong Hou, George Vulov, Daniel Quinlan, David Jefferson, Richard Fujimoto, and Richard Vuduc. A new method for program inversion. In *Proceedings of the 21st International Conference on Compiler Construction*, volume 7210 of *CC'12*, pages 81–100. Springer-Verlag, 2012.
12. Marta Cialdea Mayer and Fiora Pirri. Abduction is not deduction-in-reverse. *Logic Journal of the IGPL*, 4(1):95–108, 1996.
13. C. S. Peirce. *Collected Papers of Charles Sanders Peirce*, volume 2. Harvard University Press, 1931-1958.
14. Ira S. Pohl. Bi-directional search. *Machine Intelligence*, 6:127–140, 1971.
15. Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Leslie De Koninck. As time goes by: Constraint Handling Rules – A survey of CHR research between 1998 and 2007. *Theory and Practice of Logic Programming*, pages 1–47, 2010.
16. Tetsuo Yokoyama. Reversible computation and reversible programming languages. *Electronic Notes in Theoretical Computer Science*, 253(6):71 – 81, 2010. Proceedings of the Workshop on Reversible Computation (RC 2009).
17. Amira Zaki, Thom W. Frühwirth, and Slim Abdennadher. Towards inverse execution of constraint handling rules. *Theory and Practice of Logic Programming*, 13(4-5-Online-Supplement), 2013.