Towards Inverse Execution of Constraint Handling Rules

AMIRA ZAKI[1], THOM FRÜHWIRTH[1], SLIM ABDENNADHER[2]
[1]*Ulm University, Germany;* [2]*German University in Cairo, Egypt*
(*e-mail:* {amira.zaki,thom.fruehwirth}@uni-ulm.de, slim.abdennadher@guc.edu.eg)

Abstract

Inverse execution is a non-deterministic process of discovering the inputs to a program starting from its output. This paper deals with the inverse execution of Constraint Handling Rules (CHR). First a simple inversion technique is proposed, which produces a reverse program consisting of rules with exchanged left and right hand sides. The limitations of this method are presented and reveal the need for a different CHR execution strategy. Then an extension is provided to execute the inverse program whilst exploring all possible execution paths incrementally and exhaustively through different methods such as breadth-first traversal or random path choice. An on-line tool was implemented to transform any forward CHR program into its equivalent inverse; examples and results highlight the validity of this work.
KEYWORDS:Backwards, Constraint Handling Rules, Exhaustive Execution, Inverse, Reverse

1 Introduction

Program inversion is the backwards execution of a program, uncovering the possible inputs that generate a particular output. The process is non-deterministic, since there maybe several possible inputs for a given output. Some program inversions make very common inverse pairs, such as encryption/decryption, compression/decompression, inverting arithmetic functions, insert/delete operations on data structures, and roll-back transactions. From a software development point of view, automatic program inversion could potentially allow programmers to write only one of each inverse pair, thus the time required to write, maintain and debug such code could be halved. Moreover, backwards computation can be seen as a form of abduction, which attempts to infer the initial query from an observed result. On this basis, it is possible to have various applications based on such abductive reasoning.

Constraint Handling Rules (CHR) is a committed-choice rule-based programming language based on multi-headed multi-set guarded rewrite rules (Frühwirth 2009). It was originally designed for the special purpose of creating user-defined constraint solvers to a host-language. However over time, CHR matured to become a powerful and elegant general-purpose language with various application domains (Frühwirth et al. 1996). The traditional execution of CHR involves starting from an initial state or query and applying the program rules exhaustively until a fixed point is reached. The final state is one in which no more rules are applicable. The rule application strategy depends on the operational semantics of the language used.

A backwards or inverse execution of a CHR program, entails computing from a result to one or all predecessor states. This process is non-deterministic; various intermediate states are possible inputs which produce the same desired output. The aim of inverse execution would be to produce a maximal remote ancestor state and all other intermediate states that could yield a particular

output. The exhaustive backwards rule application and exploration of different program paths could be attainable by the use of CHR with disjunction (CHR$^{\vee}$)(Abdennadher and Schütz 1998). This extension to CHR makes use of a backtracking search to allow the invocation of different rule bodies and hence could be used for the inverse execution.

Inverse execution of CHR can lead to various research possibilities, such as abductive reasoning (Christiansen 2009), simulating CHR executions, computing tautologies starting backwards from true, computing inconsistency failures in a constraint solver by starting backwards from failure, and computing with bidirectional rules in both directions which would bring CHR more closer to its most abstract first-order logic semantics. Furthermore, an inversion could be used for a result-directed execution of CHR (Sneyers 2010).

A naive implementation of inverse execution of CHR would be to exchange the left and right hand sides of the rules. In this work, this method is presented as the simple inversion technique and illustrative examples are provided. With experiments, it becomes evident that the simple method is not sufficient to obtain the initial goal of reaching the maximal remote ancestor and all intermediate states. Maximal remote ancestors in the forwards program correspond to exhaustive rule application in the reverse program. Thus, a refined method known as exhaustive inversion is described and running examples are provided to highlight the workings and results. An online inverter tool was implemented using SWI-Prolog (Wielemaker et al. 2012), which allow users to invert CHR programs as discussed in this paper; the tool can be found at: `http://chr.informatik.uni-ulm.de/inverter/`.

The paper continues by presenting some preliminary background in Sections 2 and 3. Then the inverse semantics of CHR and its simple execution are discussed in Section 4. This is followed by an exhaustive execution mechanism for the inversion in Section 5. Finally the paper ends with some concluding remarks in Section 6.

2 Related Work

Different approaches of program inversion have been investigated for various programming paradigms. Functional languages have been a paradigm of many work, due to their high relation with mathematical functions. (Korf 1981) presented a technique for generating the inverse function of a program written in a minimal subset of LISP. It presented a running example of the inversion of a list. The language subset includes `car, cdr, cons, cond` and `equal`. Each primitive program construct becomes inverted into a separate rule, moreover additional rules are added to handle recursive and auxiliary function calls. Program constructs are easily inverted if they are invertible functions (such as `cons`). Other non-invertible functions (such as `cdr, car`) are inverted by the introduction of new variables, whose values can be sometimes determined by solving the multiple simultaneous equations obtained for them.

The inverse computation of a first-order functional language can be performed by the universal resolving algorithm (Abramov and Glück 2000; Abramov and Glück 2002). The algorithm has three main steps. First a perfect process tree containing all possible computation paths is developed for the program with a partially specified input. Every node in the tree represents a state of the program encoded within a configuration that includes a program term, an environment of variable bindings and set of restrictions on the domain of the variables. Then a forward trace is performed to generate a table of all input-output pairs from the perfect process tree, which is done by following all the paths from the root to every terminal node. Given that all splits performed in the tree are perfect, then the partitions performed for the input class are disjoint meaning that

the computation is deterministic. The last step of the algorithm involves extracting the inverse of a particular output. This is done by matching the given output against those in the input-output table and hence obtaining the possible inputs.

A similar tree-based inversion technique was that of (Matsuda et al. 2010) which uses a grammar-based approach to obtain the inverse of a program. The work states that every program has a grammar, whose complexity characterizes how difficult it is to invert it. An inverse can be derived by parsing the given output with respect to the corresponding unambiguous grammar. A production tree is obtained from the output by parsing the grammar. Then according to the correspondence, an evaluation tree of the program is generated from the production tree. The environment used is reconstructed using the evaluation tree, hence recovering the input arguments. The paper presents the grammar-based inversion with the class of regular tree grammars as the presented case study, however the framework can be extended to other classes as well.

Declarative logic programs inherently support inversion (Sickel 1978). Relations are represented by predicates, where the arguments can represent the input and output parameters of a function. The relations are defined declaratively, such that a function and its inverse are computable by the same logic program. In this work, Prolog is used which implies that all built-in predicates used would be reversible by nature.

(Shoham and McDermott 1984) presented three basic algorithms for function inversion, implemented in Prolog but can be applied to any programming language with backtracking. The algorithms are extensions of one another; they aim to inverse functions by interchanging the input and output, thereby inverting the direction of knowledge. The algorithms involve reversing the bodies of the encountered clauses and inverting basic arithmetic operations in an ad-hoc manner. The paper included as an example the inversion of list sorting. However it was shown that this technique cannot be used to invert all relations. Moreover an algorithm for redirection of predicates using a breadth-first search of the computation tree was presented, however it was demonstrated to be complete but impractical.

Most of the work on inverting imperative programs has focused on an incremental inverse. An inversion technique for C was devised in (Biswas and Mall 1999) for debugging purposes, capable of a bidirectional execution of a C program. The language constructs are classified into five different types, namely sequence statements, selection statements, iterative statements, unstructured statements and function call statements. Then similar to (Korf 1981), an inversion step is devised for each statement type. A trace file that stores the program states during the forward execution is created; it is required for statements that do not have a pure inverse. The trace file also incorporates other information such as symbol tables and other needed references for iterative and unstructured statements. Another imperative inversion attempt is presented in (Kanade et al. 2010), which was motivated to use program inversion for the representation dependence testing of algorithms. An approach is devised to invert imperative programs by synthesizing an inverse program that has a control-flow structure isomorphic to the original program using local inversion (through symbolic equation solving) to invert loop-free code fragments. The inversion is done by modifying the control flow of the input program with some reordering of statements. It works for normalization programs (mapping from one value to another) including iterative programs with arrays. Unknown variables get random assignments, then a constraint solver is invoked that tries to find satisfying assignments. A similar approach was presented in (Hou et al. 2012) for programs with arbitrary control flow and basic operations, which changed the inversion task to a graph search problem. It operates by constructing a value search graph that represents recover-

ability relationships between variable values. Special forward and backwards code is generated for any program, which guides the execution in either direction.

The series of work presented in (Nishida et al. 2005; Nishida et al. 2007; Nishida and Vidal 2011) provides a two-phase inverse compiler that produces the inverse of term rewriting systems (TRS). The first phase involves first classifying the given variables into givens and unknowns. Then for every TRS rule an inverse rule is produced by swapping the right and left hand sides with respect to the unknowns; conditionals are introduced for the new variables. Extra rules are also introduced until all conditional parts are deterministic. The second phase involves performing unraveling for the generated conditional TRSs, which means introducing new rules with fresh variables for every conditional part. Amongst the examples presented in this work set is the reverse of a list, which is also inverted in this paper.

With the exception of TRS, the operational semantics of all other languages reviewed is quite different than that of Constraint Handling Rules (CHR). In fact, CHR is quite similar to conditional TRS; however CHR contains more phenomena like the existence of global knowledge through the built-in constraint store and local variables (Abdennadher et al. 1996). Moreover propagation rules in CHR cannot be directly expressed in conditional TRS without introducing non-termination (Abdennadher 1997).

3 Constraint Handling Rules

3.1 Syntax

Constraint Handling Rules (CHR) (Frühwirth 2009; Frühwirth and Raiser 2011) is a high-level, committed-choice, constraint logic programming language. It consists of guarded rules that perform conditional transformation of multi-sets of constraints, known as a constraint store, until a fixed point is reached. It utilizes built-in constraints which are predefined by the host language, and other user-defined CHR constraints. User-defined constraints are defined by a functor/arity pair, for example `a/2` is a binary constraint of name `a`. A generalized CHR simpagation rule is given as:

$$\textit{rule-id} \ @ \ H_k \setminus H_r \ \Leftrightarrow \ G \mid B_b, B_c$$

where H_k is known as the kept head constraints and H_r as removed head constraints; both are a conjunction of one or more CHR constraints. The rule consists of an optional conjunction of host language constraints known as the guard and given by G. Following the partitioning $\mid$ is the body of the rule, and it consists of built-in constraints (B_b) and user-defined CHR constraints (B_c). Every rule has an optional unique identifier preceding it given by *rule_id*.

Two other types of rules exist which are special cases of the generalized simpagation rule, namely simplification and propagation rules. The first having no kept head constraints simplifying the head to the body, and the latter having no removed head constraints thereby just adding constraints to the store, and are of the forms:

$$\begin{aligned} \textit{simpf-rule-id} \ @ \ H_r \ &\Leftrightarrow \ G \mid B_b, B_c \\ \textit{prop-rule-id} \ \ @ \ H_k \ &\Rightarrow \ G \mid B_b, B_c \end{aligned}$$

3.2 Very Abstract Semantics

The very abstract semantics (ω_{va}) of CHR is formulated as a state transition system, where a transition corresponds to a rule application. States are goals, consisting of a conjunction of built-

in and CHR constraints. An initial state is an arbitrary one and a final state is a terminal one where no further transitions are possible. The commutative and associative properties hold for the logical conjunctions of constraints in a state, enabling permutations of conjuncts.

Let P be a CHR program and CT be a constraint theory for the built-in constraints defined by the host-language. The body of a rule B consists of both built-in constraints B_b and CHR constraints B_c, moreover H_k, H_r are a conjunction of CHR constraints, G is a conjunction of built-in constraints and C is a conjunction of both types of constraints. The non-deterministic ω_{va} semantics includes one transition rule, namely:

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:

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

Most implementations of CHR do not use this semantics as it is highly non-deterministic (Duck et al. 2004). In contrast, compilers use the refined operational semantics which defines the order of constraint execution and rule application.

3.3 Constraint Handling Rules with Disjunction

An extension to CHR which makes use of disjunction within the bodies of the CHR rules is known as CHR$^{\vee}$ (Abdennadher and Schütz 1998). It incorporates the use of a backtracking search to allow the invocation of different rule bodies. In the Prolog implementation of CHR, the use of Prolog's disjunction can be used in the body of the rules. Application of a transition rule, thus generates a branching derivation. A generalized simpagation rule with two possible disjunctive body constraints $(B_1 \vee B_2)$ is of the form:

$$\textit{simpgV-rule-id} \ @ \ H_k \setminus H_r \ \Leftrightarrow \ G \mid B_1 \,;\, B_2$$

The extended transition system for CHR$^{\vee}$ operates on a disjunction of CHR states known as a configuration: $s_1 \vee s_2 \vee \ldots \vee s_n$. The original apply transition is applicable on 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

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

---

4 Simple Inversion

This section includes an initial simple and naive attempt of the inversion of CHR. First the formal semantics of the inversion of CHR is defined. Then a simulation of the inversion is described whilst using the apply semantics, followed by illustrative examples at the end of the section. CHR (Prolog) is used, thus the inversion of built-in constraints is ensured through the reversible declarative nature of Prolog.

4.1 Backwards Semantics

The apply transition of the very abstract semantics of CHR describes a forward rule application on an initial state s_i to a final state s_f. An inversion of this transition would be to define a transition that transforms a final s_f state to an initial s_i state. This transition is known as *backwards*, and it is typically the same as the apply transition but with exchanging the left and right hand side states of the transition.

Backwards

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

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

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

It is sometimes preferred to distinguish between types of body constraints; the relevance of this will be explained later. Thus generic body constraints B can be rewritten as B_b and B_c. It follows that the initial state of the backwards transition is rewritten as $(H_k \wedge G \wedge B_b \wedge B_c \wedge C)$.

4.2 Simulating Backwards

In this work the aim is to simulate the backwards transition using the abstract CHR semantics. In other words, the result of the backwards transition will be achieved by only the apply transition. The desired transition is the following:

$$(H_k \wedge G \wedge B_b \wedge B_c \wedge C) \mapsto_{apply} (H_k \wedge H_r \wedge C)$$

However if the transition is applied using the rule r, then the result obtained is not as desired. Thus for this simulation transition to work, a reordering is necessitated in the rule applied. Following intuition and the manner of most inversion work on term-rewriting systems (Nishida et al. 2007), an inverse rule is generated by exchanging its left and right hand sides.

For simplification rules with only user constraints ($H_r \ \Leftrightarrow \ G \mid B_c$), an inverse rule would be:

$$\textit{inv-simpf} \ @ \ B_c \ \Leftrightarrow \ G \mid H_r$$

If the body of the forward rule contains built-in constraints, these cannot be simply placed on the left hand side of the inverse rule due to the syntax of CHR. These built-in body constraints can be matched in the guard of the rule. Hence the inverse of a simplification rule ($H_r \ \Leftrightarrow \ G \mid B_b, B_c$) is more correctly expressed as:

$$\textit{inv-simpf} \ @ \ B_c \ \Leftrightarrow \ B_b, G \mid H_r$$

However with propagation rules, it is not possible to simply invert the rule sides. A simple example to illustrate this with user-defined constraints `a/0` and `b/0`:

```
forward  @ a ==> b.
```

An initial state containing `(a)` produces a state `(a,b)`. However an inverse program consisting of the exchanged rule below, given a state `(a,b)` would result in the state `(a,b,a)` which is not the initial state of the forward program.

```
incorrect-inverse @ b ==> a.
```

Thus a correct inverse of propagation rules can be achieved by a simpagation rule that removes

the added constraints whilst retaining the left hand side ones. For this example the correct inverse rule that uncovers the input `(a)` would be:

```
inverse @ a \ b <=> true.
```

Therefore the generic inverse of a propagation rule can be given as:

$$\textit{inv-prop} \mathbin{@} H_k \setminus B_c \Leftrightarrow B_b, G \mid \texttt{true}$$

A simpagation rule is in fact a combination of both types of CHR rules. Thereby a combined inverse r' of a generalized simpagation rule is expressed as:

$$\textit{inv-simpg} \mathbin{@} H_k \setminus B_c \Leftrightarrow B_b, G \mid H_r$$

The employment of this inverse rule *inv-simpg* (renamed as r') with the apply transition on the initial state of the backwards transition yields the following:

$(H_k \wedge G \wedge B_b \wedge B_c \wedge C)$
$\quad \mapsto^{r'}_{apply} (H_k \wedge H_r \wedge G \wedge B_b \wedge G \wedge B_b \wedge C)$

This resultant state can be reduced by idempotency of built-in constraints to the following state:

$\equiv (H_k \wedge G \wedge H_r \wedge B_b \wedge C)$

The initial input state was $(H_k \wedge H_r \wedge G)$ and that recovered by the backwards simulation is $(H_k \wedge H_r \wedge G \wedge B_b \wedge C)$. It is easily noticeable that these two states are similar yet not the same. The result of the simulation is more strict, containing more built-in constraints which could not be removed or forgotten.

Similarly, the inversion of a generalized forward simpagation rule with a disjunctive body would result in an inverse rule with disjunctive heads. This would be represented with two inverse rules, one for every disjunct. Hence a forward rule of the form:

$$\textit{simpgV} \mathbin{@} H_k \setminus H_r \Leftrightarrow G \mid (B_{1b}, B_{1c}) \,;\, (B_{2b}, B_{2c})$$

Exchanging the left and right hand sides of the rule, then splitting the disjunction of the heads, results in the two rules:

$$\textit{inv1-simpgV} \mathbin{@} H_k \setminus B_{1c} \Leftrightarrow B_{1b}, G \mid H_r$$
$$\textit{inv2-simpgV} \mathbin{@} H_k \setminus B_{2c} \Leftrightarrow B_{2b}, G \mid H_r$$

4.3 Examples

In this subsection, the aim is to realize the simulation of the backwards transition and show some examples. The inverse program is executed under a normal CHR executor system; which is committed-choice, and executes rules top to bottom and explores goals left to right.

The process involves producing a reverse program which contains a rearranged inverse rule, as described earlier, for every forward generalized simpagation rule as shown below.

$$\textit{inv-rule-id} \mathbin{@} H_k \setminus B_c \Leftrightarrow B_b, G \mid H_r$$

If a forward simplification rule contains no CHR constraints in its body, then its inverse rule would have no constraints on the left hand side. Implementation wise, this can be resolved by the introduction of a dummy CHR constraint `top/0` to the body of the forward rule, which results in a valid inverse rule (usage will be shown in Example 4 in Subsection 5.2).

Example 1 *(List reverse)* – The reverse of a list is a typical problem whose inversion was attempted in literature (Korf 1981; Nishida and Vidal 2011). It is a one-to-one function, which can be easily inverted. A forward CHR program that performs the reversal of a list provided as a constraint `reverse/1` into a reversed list wrapped in a constraint `out/1`, can be given as:

```
rule1 @ reverse(X)         <=> reverse(X,[]).
rule2 @ reverse([X|Xs],Ys) <=> reverse(Xs,[X|Ys]).
rule3 @ reverse([],X)      <=> out(X).
```

Applying the method described would produce the following inverse program:

```
inv-rule1 @ reverse(X,[])      <=> reverse(X).
inv-rule2 @ reverse(Xs,[X|Ys]) <=> reverse([X|Xs],Ys).
inv-rule3 @ out(X)             <=> reverse([],X).
```

A sample run of a forward query `reverse([1,2,3])` produces the result `out([3,2,1])`. Then the result query `out([3,2,1])` yields `reverse([3,2,1])` with the inverse program.

A similar and more practical example is that of encrypting a list of elements using a Ceaser cipher. The encryption program is written which maps list elements to their ciphers. The inverse of this program would automatically decrypt a cipher to its original plain text. Due to limitation of space, this application is available in the on-line inversion tool implemented.

Example 2 *(Exchange sort)* – The inversion of list sorting was investigated in (Shoham and McDermott 1984) with the quick-sort algorithm. A typical CHR sort is the exchange sort, which sorts numbers stored as constraints of the form `a(Index,Value)` by exchanging any pair of elements that are in the incorrect order. This can be done with a single simplification rule:

```
esort @ a(I,V), a(J,W) <=> I>J, V<W | a(I,W), a(J,V).
```

The inverse rule becomes:

```
inv-esort @ a(I,W), a(J,V) <=> I>J, V<W | a(I,V), a(J,W).
```

A run of the forward program for a query `a(0,6),a(1,2),a(2,4)`, would yield the result `a(0,2),a(1,4),a(2,6)`. A run of the inverse program for the previously generated result of `a(0,2),a(1,4),a(2,6)`, would produce a result `a(0,6),a(1,4),a(2,2)`.

It is evident that the input uncovered by the inverse program is a possible solution but not the exact one used in the forward run. In fact, the result shows the numbers in a descending order; this is the worst-case scenario of the exchange sorting problem. However the inversion of a list sorting algorithm, should produce all permutations of the list. Any permutation would produce the same sorted result, it would be impossible to determine which one is the particular input of the forward run. The sorting problem is in fact a many-to-one problem, hence its inverse should be a one-to-many function. It is noticed that simple inversion is not sufficient for this inversion. The reason being that the compiler execution strategy of CHR, invokes a committed-choice top to bottom rule application, whilst exploring goals from left to right. This execution generates only one single result, rather than uncovering the multiple possible inputs.

5 Exhaustive Inversion

As seen by the sorting example, the normal execution of inverse rules does not suffice to uncover all possible inputs. The inversion of many-to-one functions (specially if their definition is of many rules) is clearly non-deterministic; requiring a different execution strategy capable of exploring all possible execution paths. In this work, the inverse programs are transformed to allow for an exhaustive execution. The details of the exhaustive execution is provided in Subsection 5.1. Then this transformation is applied on the simple inversion rules and illustrative examples are depicted in Subsection 5.2.

5.1 Exhaustive Execution

The operational semantics of the CHR compiler defines a committed-choice and depth-first exploration of the goals. A breadth-first execution strategy was implemented in (De Koninck et al. 2006) as a source-to-source transformation of an initial program to one with nodes and edges, allowing different traversal strategies. In this work a different transformation is used, inspired from conflict resolution (Frühwirth 2009). The transformation is adopted to facilitate the exploration of different execution schemes for CHR. The original idea presented in the book is extended with disjunction, such that every intermediate node in the traversal is a possible solution.

The transformation works by delaying the execution of the rule bodies. All rules with head constraints that match the goal are collected in a rule-set list (using `delay-ignore` rules). A rule (named `collect`) merges these lists, such that for any goal there is a set of all applicable rules. Then once a rule is chosen from within a rule-set (using `halt-fire` and `halt-choose` rules), the chosen rule is fired by applying its body (through its respective `apply` rule). All intermediate goals are obtained using CHR$^\vee$ by introducing a disjunctive `true` in appropriate locations.

The left-hand side of a CHR rule which is needed to check for matching is encapsulated in a `rule/2` constraint containing lists of the heads kept and removed respectively. An additional global constraint `strategy/1` is used to decide on the execution strategy followed. `ruleset` is a constraint used to collect all the applicable CHR rules in its argument list. `fire` is an auxiliary constraint to trigger the start of the execution, and `apply` is a constraint that executes a chosen rule by triggering the appropriate apply rule.

A normal execution would entail that any rule whose heads match would be a possible rule and added to the rule-set. However, an exhaustive execution is desired which should consider incomplete rule-sets. Hence this is achieved by a `delay-ignore` rule, which either adds the current rule to the rule-set, or ignores this program rule.

Thus for every simpagation rule ($H_k \setminus H_r \Leftrightarrow G \mid B_b, B_c$), a pair of rules is introduced (one to add/not-add the rule to the rule-set and the other in case the rule is chosen for application):

```
delay-ignore @ Hk, Hr ⇒ G | ruleset([rule(Hk,Hr)]); true.
apply @ Hk \ Hr, apply(rule(Hk,Hr)) ⇔ G | B.
```

To merge two rule-sets into one constraint, a generic rule is added to any translated program:

```
collect @ ruleset(L1), ruleset(L2) <=> append(L1,L2,L3), ruleset(L3).
```

During the execution of the rules contained in a rule-set, it is possible to either decide that the current state is a solution, or that the rule-set is to be fired (through a local `fireruleset` constraint). This is accomplished through the `halt-fire` rule.

```
halt-fire   @ ruleset(L) \ fire <=> L\=[] | true; fireruleset.
```

Once the rule-set is to be executed, a rule must be chosen from the rule-set list; this is determined by adding a `choose/3`. Then a chosen rule, might be either applied and the resultant state be an intermediate state or that further application of the rule-set should be performed. This necessitates the rule `halt-choose` shown below:

```
halt-choose @ fireruleset, ruleset(L)
    <=> choose(L,R,L1), ruleset(L1), apply(R), (true; fireruleset).
```

The choice of a rule from a rule-set depends on the `strategy(`S`)` constraint. Thus to implement a breadth-first-traversal (where $S = bfs$) which extracts the first rule `R` from the rule-set `L` with the remaining rules in `L1`, the following rule can be defined:

```
bfs @ strategy(S) \ choose(L,R,L1) <=> S = bfs | L=[R|L1].
```

For a random execution of the rules (where $S = rand$), and given a predicate `random_select(R, L, L1)` (Wielemaker et al. 2012) which selects a random rule `R` from `L` with the remaining rules in `L1`, then the following rule can be defined:

```
rand @ strategy(S) \ choose(L,R,L1) <=> S = rand | random_select(R,L,L1).
```

5.2 Implementation of Exhaustive Inverse

The exhaustive exploration of the inverse of a CHR program can be performed by combining the simple inversion with the exhaustive execution transformation.

Step 1 - Exhaustive Version of the Inverse – Every forward rule `r` is transformed into two rules, `delay-ignore-r` and `apply-r`. If one is to substitute the inverse rule constraints into the exhaustive rules, then for any forward rule the following pair can be generated:

`delay-ignore-r @` $H_k, B_c \Rightarrow B_b, G$ `| ruleset([rule(`H_k, B_c`)]); true.`
`apply-r @` H_k `\` B_c`, apply(rule(`H_k, B_c`)) ⇔` B_b, G `|` H_r`.`

Step 2 - Generic Rules – The generic `collect`, `halt-fire` and `halt-choose` rules, and the required strategy exploration rule e.g. `bfs` or `rand` are added to the transformed program.

During the inverse execution, guards and body built-in constraints may contain variables that are unbound yet. In this work, it is assumed that all built-in constraints used are bidirectional and an internal solver is responsible for determining the values of these variables. Moreover, the transformation is restricted to the class of range-restricted CHR rules.

Furthermore, a translator was implemented capable of transforming any ordinary CHR program into an inverse CHR program, supporting simple inversion and exhaustive inversion with breadth-first or random execution strategies. The translator can be found under: http://chr.informatik.uni-ulm.de/inverter/.

Example 3 *(Exchange sort revisited)* – Accordingly, it follows to revisit the exchange sort example and apply an exhaustive inversion with a breadth-first execution strategy ($S = bfs$). Thus for the forward `esort` rule, two rules are generated:

```
delay-ignore-esort @ a(I,W), a(J,V) ==> I>J, V<W
    | ruleset([rule([],[a(I,W),a(J,V)])]); true.
apply-esort @ a(I,W), a(J,V), apply(rule([],[a(I,W),a(J,V)]))
    <=> I>J, V<W | a(I,V),a(J,W).
```

The same sample query is repeated, but rewriting it to include the exhaustive execution trigger constraint, i.e. `strategy(bfs),a(0,2),a(1,4),a(2,6),fire`. It generates several results, which form the complete set of all permutations. However there exists several redundancies; the intensive use of disjunction produces several duplicate states which are revisited multiple times. The inverse program 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 that yield a certain sorted output is finite. Shown below are some of the possible inverses (duplicates are removed):

```
a(0,2),a(1,4),a(2,6);
a(0,6),a(2,2),a(1,4);
a(0,2),a(1,6),a(2,4);
a(0,6),a(1,4),a(2,2);
a(0,4),a(1,6),a(2,2);  ...
```

Example 3 *(Greatest Common Divisor)* – The Euclidean algorithm for the computation of the greatest common divisor for a multi-set of numbers represented by `gcd/1` constraints, is expressed in CHR as shown below (where `eq/2` is a built-in equality constraint). This is a multi-line program, hence the rule order of the inverse program would be interesting for observation.

```
base @ gcd(0) <=> true.
gcd  @ gcd(N) \ gcd(M) <=> 0<N, N=<M | L eq M-N, gcd(L).
```

The base case is a simplification rule with a body consisting of only built-in constraints and requires special handling as explained earlier. The second rule is a normal simpagation rule. Exhaustive inversion transformation ($S = bfs$) is applied, hence the forward program is transformed into the following four rules:

```
delay-ignore-base @ top ==> ruleset([rule([],[top])]); true.
apply-base @ top, apply(rule([],[top])) <=> gcd(0), top.
delay-ignore-gcd @ gcd(N), gcd(L)
   ==> L eq M-N, 0<N, N=<M | ruleset([rule([gcd(N)],[gcd(L)])]); true.
apply-gcd @ gcd(N) \ gcd(L),apply(rule([gcd(N)],[gcd(L)]))
   <=> L eq M-N, 0<N, N=<M | gcd(M).
```

A typical run of the forward program to compute the greatest common divisor of three numbers is given as:

```
?- gcd(12),gcd(8),gcd(4).
gcd(4).
```

Using the simple inversion method on an input query of `gcd(4),top`, results in an infinite running depth-first traversal of the second inverse rule. Although no terminating result can be achieved, an internal trace of the backwards search path followed would be:

```
?- gcd(4),top.
--> gcd(4),gcd(0) --> gcd(4),gcd(4) --> gcd(4),gcd(8) --> gcd(4),gcd(12) --> ...
```

Using a breadth-first execution strategy and taking the result of the forward run as an input for the inverse program, yields the output depicted below *(some lines have been trimmed to reduce space)*.

```
?- strategy(bfs),gcd(4),top,fire.
 gcd(4);
 gcd(0),gcd(4);
 gcd(4),gcd(4);
 gcd(0),gcd(4),gcd(4);
 gcd(8),gcd(0),gcd(4);  ...
```

If a random execution strategy is used, it would allow the exploration of different possible inverse execution paths, without being confined in a depth-first or breadth-first manner. Starting from a query of one number, there are infinitely many possible inputs. Repeating the previous query produces the different runs below, which give various different input queries with varying number of input constraints:

```
?- strategy(rand),gcd(4),top,fire.
 gcd(4);
 gcd(0),gcd(4);
 gcd(4),gcd(4);
 gcd(8),gcd(4);
 gcd(12),gcd(8);  ...
```

```
?- strategy(rand),gcd(4),top,fire.
 gcd(4);
 gcd(0),gcd(4);
 gcd(4),gcd(4);
 gcd(8),gcd(4);
 gcd(12),gcd(4);
 gcd(0),gcd(12),gcd(4);
 gcd(4),gcd(12),gcd(4);
 gcd(0),gcd(4),gcd(12),gcd(4); ...
```

```
?- strategy(rand),gcd(4),top,fire.
 gcd(4);
 gcd(0),gcd(4);
 gcd(0),gcd(0),gcd(4);
 gcd(4),gcd(0),gcd(4);
 gcd(4),gcd(4),gcd(4); ...
```

```
?- strategy(rand),gcd(4),top,fire.
 ...
 gcd(4),gcd(8),gcd(0),gcd(4);
 gcd(8),gcd(4),gcd(8),gcd(4);
 gcd(8),gcd(8),gcd(8),gcd(4);
 gcd(0),gcd(8),gcd(8),gcd(8),gcd(4);
 gcd(8),gcd(8),gcd(8),gcd(8),gcd(4);
 ...
```

6 Conclusion

Inverse execution is the process of computing backwards the inputs of a program starting from its output. Many techniques have been researched for the inversion of functional languages, term-rewriting systems and others. In the paper, inverse execution was attempted for CHR, which is a novel model for this language.

The inversion technique devised is similar to that of term-rewriting systems (Nishida and Vidal 2011), in that the sides of the rules are exchanged. This idea is extended to accommodate for the presence of CHR propagation rules. For every forward rule, an inverse rule is produced; such that constraints removed by a forward rule are added by the inverse rule, those added by a forward rule are removed by the inverse and those kept in the forward rule remain in the inverse rule. The paper introduced two execution techniques for inverting programs, simple inversion and exhaustive inversion. The simple inversion was accomplished under a normal CHR system. The execution is performed in a depth-first manner (i.e. exploring goals left to right and rules top to bottom), which yields a correct yet incomplete result. It was found effective to inverse one-to-one functions, yet was limited to programs with only one rule.

However for many-to-one functions, an inverse would be of a one-to-many cardinality. A second execution attempt, namely exhaustive inversion, was devised to ensure that all possible inversion paths are explored hence uncovering all possible inputs. The technique is inspired from a conflict resolution implementation (Frühwirth 2009) to handle the different paths of CHR rules whilst utilizing the disjunction offered by CHR$^{\vee}$ (Abdennadher and Schütz 1998). Experiments were performed with a breadth-first execution of the inverse tree and a random execution for some extreme cases.

One-to-one functions are reversible, and the input can be obtained from a given output. On the other hand, many-to-one functions are also reversible, but it is not possible to determine which of the many inputs produced a certain output. Hence if the function has infinitely many possibilities that yield a certain output, then the inverse program is non-terminating (such as the G.C.D. problem). However, the list sorting problem is also a many-to-one function but the number of input permutations of a list is finite; thus its inverse program is a terminating one.

This work revealed that exhaustive inversion becomes a search problem, hence its complexity becomes exponential. As future work, the aim is to test the inversion approach on real experimental data to check how it scales up. The examples chosen in this paper and those included in the on-line tool are toy examples; serving as a proof of concept. However the intention is to experiment with more large scale inversion problems, like the encryption/decryption of large text files and further applications. The examples presented show that inversion might provide interesting insights on problems from a different viewpoint; such as the observation that the simple inverse of sorting yields the worst-case list, or that the inversion of the greatest common divisor of a number yields a tree of all possible sets of numbers generating that divisor.

The exhaustive execution strategy used can be optimized by enhancing the use of disjunction in the exhaustive execution and investigating other possible execution strategies. This future optimization could reduce the number of redundancies obtained by revisiting some inverse states. Alternatively, it could be valuable to substitute the exhaustive execution with that of the angelic semantics (Martinez 2011). This semantics aims to explore all possible logical consequences, to obtain a complete set of computed goals for an initial goal, however an operational semantics for this work has not been published to date.

References

ABDENNADHER, S. 1997. Operational semantics and confluence of constraint propagation rules. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming*. 252–266.

ABDENNADHER, S., FRÜHWIRTH, T., AND MEUSS, H. 1996. On confluence of constraint handling rules. In *Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming*. 1–15.

ABDENNADHER, S. AND SCHÜTZ, H. 1998. CHR$^{\vee}$: A flexible query language. In *Flexible Query Answering Systems*. Lecture Notes in Computer Science, vol. 1495. Springer-Verlag, 1–14.

ABRAMOV, S. AND GLÜCK, R. 2000. The universal resolving algorithm: Inverse computation in a functional language. In *Mathematics of Program Construction*. 187–212.

ABRAMOV, S. AND GLÜCK, R. 2002. Principles of inverse computation and the universal resolving algorithm. In *The Essence of Computation*. Lecture Notes in Computer Science, vol. 2566. Springer Berlin Heidelberg, 269–295.

BISWAS, B. AND MALL, R. 1999. Reverse execution of programs. *SIGPLAN Notices 34,* 4, 61–69.

CHRISTIANSEN, H. 2009. Executable specifications for hypothesis-based reasoning with prolog and constraint handling rules. *J. Applied Logic 7,* 3, 341–362.

DE KONINCK, L., SCHRIJVERS, T., AND DEMOEN, B. 2006. Search strategies in CHR(Prolog). In *Proceedings of the 3rd Workshop on Constraint Handling Rule*, T. Schrijvers and T. Frühwirth, Eds. K.U.Leuven, Department of Computer Science, Technical report CW 452, 109–124.

DUCK, G. J., STUCKEY, P. J., GARCÍA DE LA BANDA, M., AND HOLZBAUR, C. 2004. The refined operational semantics of Constraint Handling Rules. In *Proceedings of the 20th International Conference on Logic Programming*, B. Demoen and V. Lifschitz, Eds. 90–104.

FRÜHWIRTH, T. 2009. *Constraint Handling Rules*. Cambridge University Press.

FRÜHWIRTH, T., BRISSET, P., AND MOLWITZ, J.-R. 1996. Planning cordless business communication systems. *IEEE Expert: Intelligent Systems and Their Applications 11,* 1, 50–55.

FRÜHWIRTH, T. AND RAISER, F., Eds. 2011. *Constraint Handling Rules: Compilation, Execution, and Analysis*. Books on Demand.

HOU, C., VULOV, G., QUINLAN, D., JEFFERSON, D., FUJIMOTO, R., AND VUDUC, R. 2012. A new method for program inversion. In *Proceedings of the 21st International Conference on Compiler Construction*. CC'12, vol. 7210. Springer-Verlag, 81–100.

KANADE, A., ALUR, R., RAJAMANI, S., AND RAMANLINGAM, G. 2010. Representation dependence testing using program inversion. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE '10. 277–286.

KORF, R. E. 1981. Inversion of applicative programs. In *Proceedings of the 7th International Joint Conference on Artificial intelligence*. IJCAI'81. 1007–1009.

MARTINEZ, T. 2011. Angelic CHR. In *Proceedings of the 8th Workshop on Constraint Handling Rules*. CHR'11. GUC, Technical report, 19–31.

MATSUDA, K., MU, S.-C., HU, Z., AND TAKEICHI, M. 2010. A grammar-based approach to invertible programs. In *Proceedings of the 19th European Conference on Programming Languages and Systems*. ESOP'10. 448–467.

NISHIDA, N., SAKAI, M., AND KATO, T. 2007. Convergent term rewriting systems for inverse computation of injective functions. In *Proceedings of the 9th International Workshop on Termination*. WST'07. 77–81.

NISHIDA, N., SAKAI, M., AND SAKABE, T. 2005. Partial inversion of constructor term rewriting systems. In *Proceedings of the 16th International Conference on Term Rewriting and Applications*. RTA'05. 264–278.

NISHIDA, N. AND VIDAL, G. 2011. Program inversion for tail recursive functions. In *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications*. LIPIcs. 283–298.

SHOHAM, Y. AND MCDERMOTT, D. V. 1984. Knowledge inversion. In *Proceedings of the National Conference on Artificial Intelligence*. AAAI. 295–299.

SICKEL, S. 1978. Invertibility of logic programs. Tech. rep., California Univ Santa Cruz Information Sciences. August.

SNEYERS, J. 2010. Result-directed CHR execution. In *Proceedings of the 7th Workshop on Constraint Handling Rules*. CHR'07. 25–31.

WIELEMAKER, J., FRÜHWIRTH, T., KONINCK, L. D., TRISKA, M., AND UNESON, M. 2012. *SWI Prolog Reference Manual 6.2.2*. Books on Demand.