

Satisfiability of Constrained Horn Clauses on Algebraic Data Types: A Transformation-based Approach ^{*}

Emanuele De Angelis ¹ [0000-0002-7319-8439],
 Fabio Fioravanti ² [0000-0002-1268-7829],
 Alberto Pettorossi ^{3,1} [0000-0001-7858-4032], and
 Maurizio Proietti ¹ [0000-0003-3835-4931]

¹ IASI-CNR, Rome, Italy, {emanuele.deangelis,maurizio.proietti}@iasi.cnr.it

² DEC, University of Chieti-Pescara, Italy, fabio.fioravanti@unich.it

³ DICII, University of Rome Tor Vergata, Rome, Italy, alberto.pettorossi@uniroma2.it

Abstract. We address the problem of checking the satisfiability of Constrained Horn Clauses (CHCs) defined on Algebraic Data Types (ADTs), such as lists and trees. We propose a new technique for transforming CHCs defined on ADTs into CHCs where the arguments of the predicates have only basic types, such as integers and booleans. Thus, our technique avoids, during satisfiability checking, the explicit use of proof rules based on induction over the ADTs. The main extension over previous techniques for ADT removal is a new transformation rule, called *differential replacement*, which allows us to introduce auxiliary predicates, whose definitions correspond to lemmas that are used when making inductive proofs. We present an algorithm that performs the automatic removal of ADTs by applying the new rule, together with the traditional folding/unfolding rules. We prove that, under suitable hypotheses, the set of the transformed clauses is satisfiable if and only if so is the set of the original clauses. By an experimental evaluation, we show that the use of the new rule significantly improves the effectiveness of ADT removal. We also show that our approach is competitive with respect to tools that extend CHC solvers with the use of inductive rules.

1 Introduction

Constrained Horn Clauses (CHCs) constitute a fragment of the first order predicate calculus, where the Horn clause syntax is extended by allowing *constraints* on specific domains to occur in clause premises. CHCs have gained popularity as a logical formalism well suited for automatic verification of programs [5]. Indeed, many verification problems can be reduced to the satisfiability problem for CHCs.

^{*} This work has been partially supported by the National Group of Computing Science (GNCS-INdAM). This paper is an improved, extended version of a paper that the authors have presented at IJCAR 2020 [18] and also at the 35th Italian Conference on Computational Logic (CILC 2020).

Satisfiability of CHCs is a particular case of *Satisfiability Modulo Theories* (SMT), understood here as the general problem of determining the satisfiability of (possibly quantified) first order formulas where the interpretation of some function and predicate symbols is defined in a given constraint theory (also called *background theory*) [4]. Recent advances in the field have led to the development of a number of very powerful SMT *solvers* (and, in particular, CHC solvers), which aim at solving satisfiability problems with respect to a large variety of constraint theories. Among the SMT solvers, we would like to mention CVC4 [3], MathSAT [7], and Z3 [19], and among solvers with specialized engines for CHCs, we recall Eldarica [29], HSF [27], RAHFT [33], VeriMAP [11], and Z3-SPACER [35].

Even if SMT algorithms for unrestricted first order formulas suffer from incompleteness limitations due to general undecidability results, most of the above mentioned tools work well in practice when acting on constraint theories, such as Booleans, Uninterpreted Function Symbols, Linear Integer or Real Arithmetic, Bit Vectors, and Arrays. However, when formulas contain universally quantified variables ranging over inductively defined *algebraic data types* (ADTs), such as lists and trees, then the SMT/CHC solvers often show poor results, as they do not incorporate induction principles for the ADT in use.

To mitigate this difficulty, some SMT/CHC solvers have been enhanced by incorporating suitable induction principles [46, 53, 54], similarly to what has been done in automated theorem provers [6]. The most creative step which is needed when extending SMT solving with induction, is the generation of the auxiliary lemmas that are required for proving the main conjecture.

An alternative approach, proposed in the context of CHCs [15], consists in transforming a given set of clauses into a new set: (i) where all ADT terms are removed (without introducing new function symbols), and (ii) whose satisfiability implies the satisfiability of the original set of clauses. This approach has the advantage of separating the concern of dealing with ADTs (which is considered at transformation time) from the concern of dealing with simpler, non-inductive constraint theories (which is considered at solving time), thus avoiding the complex interaction between inductive reasoning and constraint solving. It has been shown [15] that the transformational approach compares well with induction-based solvers if lemmas are not needed in the proofs. However, in some satisfiability problems, if suitable lemmas are not provided, the transformation fails to remove the ADT terms.

The main contributions of this paper are as follows.

- (1) We extend the transformational approach by proposing a new rule, called *differential replacement*, based on the introduction of appropriate *difference predicates*, which play a role similar to that of lemmas in inductive proofs. We prove that the combined use of the fold/unfold transformation rules [23] and the differential replacement rule is *sound*, that is, if the transformed set of clauses is satisfiable, then the original set of clauses is satisfiable. We also study some sufficient conditions that guarantee that the use of those rules

is *sound and complete* in the sense that the transformed set of clauses is satisfiable if and only if the original set of clauses is satisfiable.

- (2) We develop a transformation algorithm that removes ADTs from CHCs by applying the fold/unfold and the differential replacement rules in a fully automatic way.
- (3) Due to the undecidability of the satisfiability problem for CHCs, in general our technique for ADT removal may not terminate. Thus, we evaluate its effectiveness from an experimental point of view and, in particular, we discuss the results obtained by the implementation of our technique in a tool, called ADTREM. We consider a set of CHC satisfiability problems on ADTs taken from various benchmarks which are used for evaluating inductive theorem provers. The experiments show that ADTREM is competitive with respect to Reynolds and Kuncak’s tool that augments the CVC4 solver with inductive reasoning [46].

The paper is structured as follows. In Section 2 we briefly present an introductory, motivating example. In Section 3 we recall some basic notions about CHCs. In Section 4 we introduce the rules used in our transformation technique and, in particular, the novel differential replacement rule, and we show the soundness of the rules we consider. In Section 5 we present a transformation algorithm, called \mathcal{R} , that uses the transformation rules for removing ADTs from sets of CHCs. In Section 6 we show that, under suitable hypotheses, the transformation rules we use are also complete. In Section 8 we illustrate the ADTREM tool and we present the experimental results we have obtained. Finally, in Section 9 we discuss the related work and make a few concluding remarks.

2 A Motivating Example

Let us consider the following functional program *Reverse*, which we write using the OCaml syntax [38]:

```

type list = Nil | Cons of int * list;;
let rec append l ys = match l with
| Nil -> ys          | Cons(x,xs) -> Cons(x,(append xs ys));;
let rec snoc l y = match l with
| Nil -> Cons(y,Nil) | Cons(x,xs) -> Cons(x,snoc xs y);;
let rec reverse l = match l with
| Nil -> Nil         | Cons(x,xs) -> snoc (reverse xs) x;;
let rec len l = match l with
| Nil -> 0           | Cons(x,xs) -> 1 + len xs;;

```

The functions `append`, `reverse`, and `len` compute list concatenation, list reversal, and list length, respectively. The function `snoc`, given a list `l` and an element `y`, returns the list obtained by inserting `y` at the end of `l`.

Suppose we want to prove the following property concerning those functions:

$$\forall xs, ys. \text{len}(\text{reverse}(\text{append } xs \text{ } ys)) = (\text{len } xs) + (\text{len } ys) \quad \text{Property (1)}$$

This property follows from the facts that: (i) by appending a list `xs` of length `n0` and a list `ys` of length `n1`, we get a list of length `n0+n1`, and (ii) by reversing a

list, we get a list with the same length. In the program *Reverse* we have assumed that the elements of the lists are integers, but Property (1) holds independently of the type of the elements of the lists. Inductive theorem provers construct a proof of Property (1) by induction on the structure of the list `l`, by assuming the knowledge of the following lemma:

$$\forall x, xs. \text{len}(\text{snoc } xs \ x) = (\text{len } xs) + 1 \quad \text{Lemma (2)}$$

which states that, given a list `xs` of length `n` and an element `x`, `snoc xs x` returns a list of length `n+1`.

The approach we follow in this paper avoids both the explicit use of induction principles and the knowledge of *ad hoc* lemmas. First, we consider the translation of Property (1) into a set of constrained Horn clauses, where, for every function f defined by a given functional program, the atom $f(X, Y)$ is the translation of ‘ $f(X)$ evaluates to Y in the *call-by-value* semantics’ [15, 53]. (Automated translation techniques have also been proposed for imperative languages with functions [13, 27].) We get set *RevCls* of the following clauses⁴:

1. `false :- N2=\=N0+N1, append(Xs,Ys,Zs), reverse(Zs,Rs), len(Xs,N0), len(Ys,N1), len(Rs,N2).`
2. `append([],Ys,Ys).`
3. `append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).`
4. `reverse([],[]).`
5. `reverse([X|Xs],Rs) :- reverse(Xs,Ts), snoc(Ts,X,Rs).`
6. `snoc([],Y,[Y]).`
7. `snoc([X|Xs],Y,[X|Zs]) :- snoc(Xs,Y,Zs).`
8. `len([],N) :- N=0.`
9. `len([X|Xs],N1) :- N1=N0+1, len(Xs,N0).`

RevCls is satisfiable if and only if Property (1) holds. However, state-of-the-art CHC solvers, such as Eldarica or Z3-SPACER, fail to prove the satisfiability of the set *RevCls* of clauses, because those solvers do not incorporate any induction principle on lists.

To overcome this difficulty, we may apply the transformational approach based on the fold/unfold rules [15], whose objective is to transform a given set of clauses into a new set without occurrences of list variables. Then, the satisfiability of the derived set of clauses can be checked by using CHC solvers based on the theory of Linear Integer Arithmetic (*LIA*) only. The soundness of the transformation rules ensures that the satisfiability of the transformed clauses implies the satisfiability of the original ones.

In the transformational approach the fold/unfold rules are applied according to a given algorithm and their ability of eliminating ADTs (and lists, in particular) very much depends on the algorithm used.

The *Elimination Algorithm*, proposed in previous work [15], allows the removal of ADT variables in many non-trivial examples. However, that algorithm is not successful in our case here because it is not able to discover auxiliary

⁴ In the examples, we use Prolog syntax for writing clauses, instead of the more verbose SMT-LIB syntax. The predicates `=\=` (different from), `=` (equal to), `<` (less-than), `>=` (greater-than-or-equal-to) denote constraints between integers.

properties, such as Lemma (2) in our case, which are often needed during the transformation. A similar limitation also applies to some tools that extend SMT solvers with induction [46, 53, 54]. Indeed, those tools sometimes fail to discover the suitable lemmas (such as Lemma (2)) that are needed for the inductive proofs.

The new ADT removal algorithm \mathcal{R} , which we present in this paper (see Section 5), extends the Elimination Algorithm by providing a technique for the automatic invention of predicates that correspond to the suitable lemmas needed for eliminating ADTs from sets of CHCs. Indeed, when applied to the set *RevCls* of clauses, Algorithm \mathcal{R} introduces three new predicates defined, respectively, by the following clauses:

```
D1. new1(N0,N1,N2) :- append(Xs,Ys,Zs), reverse(Zs,Rs), len(Xs,N0),
                      len(Ys,N1), len(Rs,N2).
D2. new2(N1,N2) :- reverse(Zs,Rs), len(Zs,N1), len(Rs,N2).
D3. diff(X,N2,N21) :- snoc(Rs,X,R1s), len(R1s,N21), len(Rs,N2).
```

Predicate `new1` is defined by taking the conjunction of the atoms occurring in the body of clause 1. Predicate `new2` is defined from the body of a clause derived by unfolding clause D1 with respect to the atom `append` (using clause 2). The definition of predicate `diff` is based on a more complex mechanism as it is derived by matching clause D1 against the following clause `D1*` obtained by unfolding clause D1 with respect to the atoms `append` (using clause 3), `reverse`, and `len`:

```
D1*. new1(N01,N1,N21) :- N01=N0+1, append(Xs,Ys,Zs), reverse(Zs,Rs),
                        len(Xs,N0), len(Ys,N1), snoc(Rs,X,R1s),
                        len(R1s,N21).
```

The definition of the predicate `diff`, given by clause D3, comes from the mismatch between the bodies of clauses D1 and `D1*` and, for this reason, that predicate is called a *difference predicate*. Indeed, the body of clause D3 is made out of: (i) the atoms `snoc(Rs,X,R1s)` and `len(R1s,N21)`, which occur in clause `D1*` and do not occur in clause D1, and (ii) the atom `len(Rs,N2)`, which occurs in clause D1 and does not occur in clause `D1*`.

In Section 5, we will provide the formal definition of Algorithm \mathcal{R} . We will also revisit the *Reverse* example and we will give a detailed account on how the definitions of the predicates `new1`, `new2`, and `diff` can be introduced in a fully automatic way.

The transformation of *RevCls* together with the additional clauses D1, D2, and D3, is now done according to a routine application of the fold/unfold rules. Indeed, we get the following final set *TransfRevCls* of clauses without list arguments (the numbering of the clauses refers to the detailed transformation shown in Section 5):

```
10. false :- N2=\=N0+N1, new1(N0,N1,N2).
15. new1(N0,N1,N2) :- N0=0, new2(N1,N2).
17. new1(N0,N1,N2) :- N0=N+1, new1(N,N1,M), diff(X,M,N2).
18. new2(M,N) :- M=0, N=0.
19. new2(M1,N1) :- M1=M+1, new2(M,N), diff(X,N,N1).
```

20. `diff(X,N0,N1) :- N0=0, N1=1.`
 21. `diff(X,N0,N1) :- N0=N+1, N1=M+1, diff(X,N,M).`

The Eldarica CHC solver proves the satisfiability of *TransfRevCls* by computing the following *LIA* model, which we write in a Prolog-like syntax as a set of constrained facts:

```
new1(N0,N1,N2) :- N2=N0+N1, N0>=0, N1>=0, N2>=0.
new2(M,N) :- M=N, M>=0, N>=0.
diff(X,N2,N21) :- N21=N2+1, N2>=0.
```

Note that, if in clause D3 we replace the atom `diff(N2,X,N21)` by its model computed by Eldarica, namely the constraint ‘`N21=N2+1, N2>=0`’, we get the following formula:

$$\forall \text{Rs}, \text{X}, \text{R1s}, \text{N21}, \text{N2}. \text{snoc}(\text{Rs}, \text{X}, \text{R1s}), \text{len}(\text{R1s}, \text{N21}), \text{len}(\text{Rs}, \text{N2}) \rightarrow \text{N21}=\text{N2}+1, \text{N2}>=0$$

which is equivalent to Lemma (2). Thus, in this case, the introduction of the difference predicate performed by Algorithm \mathcal{R} , can be viewed as a way of automatically introducing the lemma needed for constructing the inductive proof of Property (1).

3 Constrained Horn Clauses

In this section we recall some basic notions about CHCs. Let *LIA* be the theory of linear integer arithmetic and *Bool* be the theory of boolean values. A *constraint* is a quantifier-free formula of $LIA \cup Bool$. Let \mathcal{C} denote the set of all constraints. Let \mathcal{L} be a typed first order language with equality [22] which includes the language of $LIA \cup Bool$. Let *Pred* be a set of predicate symbols in \mathcal{L} not occurring in the language of $LIA \cup Bool$.

The integer and boolean types are said to be *basic types*. Here, for reasons of simplicity, we do not consider other basic types, such as real numbers, arrays, and bit-vectors, which are usually supported by SMT solvers [3, 19, 29]. The non-basic types are collectively called *algebraic data types* (ADTs), which are specified by suitable data-type declarations such as the `declare-datatypes` declarations adopted by SMT solvers.

An *atom* is a formula of the form $p(t_1, \dots, t_m)$, where p is a typed predicate symbol in *Pred*, and t_1, \dots, t_m are typed terms constructed out of individual variables, individual constants, and function symbols. A *constrained Horn clause* (or a CHC, or simply, a *clause*) is an implication of the form $H \leftarrow c, B$ (for clauses we use the logic programming notation, where comma denotes conjunction). The conclusion (or *head*) H is either an atom or *false*, the premise (or *body*) is the conjunction of a constraint $c \in \mathcal{C}$, and a (possibly empty) conjunction B of atoms. If the head H of a clause is an atom of the form $p(t_1, \dots, t_n)$, the predicate p is said to be a *head predicate*. A clause whose head is an atom is called a *definite clause*, and a clause whose head is *false* is called a *goal*.

We assume that, for every atom A occurring in a clause, (i) each term of basic type occurring in A is a variable, and (ii) no variable of basic type occurs in A more than once. For instance, the atom `p(X, [Y | T])` may occur in a clause,

while by Condition (i), the atoms $p(3, [Y \mid T])$ and $p(X, [Y+Z \mid T])$ may not. Conditions (i) and (ii) on atoms can always be enforced at the expense of introducing new variables subject to constraints in the body of the clause. These conditions ensure that, when applying the unfolding rule (see Section 4), the unification of terms of basic type can be delegated to constraint solving.

We assume that all variables in a clause are universally quantified in front, and thus we can freely rename them. Clause C is said to be a *variant* of clause D if C can be obtained from D by renaming variables and rearranging the order of the atoms in its body. Given a term t , by $vars(t)$ we denote the set of all variables occurring in t . Similarly, for the set of all variables occurring in a formula or a set of formulas. Given a formula φ in \mathcal{L} , we denote by $\forall(\varphi)$ its *universal closure*.

Let \mathbb{D} be the usual interpretation for the symbols in $LIA \cup Bool$, and let a \mathbb{D} -*interpretation* be an interpretation of \mathcal{L} that agrees with \mathbb{D} , for all symbols occurring in $LIA \cup Bool$. A \mathbb{D} -model of a clause C is a \mathbb{D} -interpretation that makes C true. A \mathbb{D} -model of a set P of clauses is a \mathbb{D} -model of every clause in P . The reference to the interpretation \mathbb{D} will be omitted when it is irrelevant or understood from the context.

A set P of CHCs is said to be \mathbb{D} -*satisfiable* (or *satisfiable*, for short) if it has a \mathbb{D} -model, and it is said to be \mathbb{D} -*unsatisfiable* (or *unsatisfiable*, for short), otherwise. Given two \mathbb{D} -interpretations \mathbb{I} and \mathbb{J} , we say that \mathbb{I} is *included* in \mathbb{J} if for all ground atoms A , $\mathbb{I} \models A$ implies $\mathbb{J} \models A$. Every set P of definite clauses is satisfiable and has a *least* (with respect to inclusion) \mathbb{D} -model, denoted $M(P)$, which is equal to the set of all ground atoms that are true in all \mathbb{D} -models of P [31]. If P is any set of constrained Horn clauses and Q is the set of the goals in P , then we define $Definite(P)$ to be the set $P \setminus Q$. It is the case that P is satisfiable if and only if $M(Definite(P)) \models Q$.

We will often use a variable as an argument of a predicate to actually denote a tuple of variables. For instance, we will write $p(X, Y)$, instead of $p(X_1, \dots, X_m, Y_1, \dots, Y_n)$, whenever the values of m (≥ 0) and n (≥ 0) are not relevant. Whenever the order of the variables is not relevant, we will feel free to identify tuples of distinct variables with finite sets.

We will also extend to finite tuples the operations and relations which are usually defined on sets. Given two tuples X and Y of distinct variables, (i) their *union* $X \cup Y$ is obtained by concatenating them and removing all duplicated occurrences of variables, (ii) their *intersection* $X \cap Y$ is obtained by removing from X the variables which do not occur in Y , (iii) their *difference* $X \setminus Y$ is obtained by removing from X the variables which occur in Y , and (iv) $X \subseteq Y$ holds if every variables of X occurs in Y . For all $m \geq 0$, equality of m -tuples of terms is defined as follows: $(u_1, \dots, u_m) = (v_1, \dots, v_m)$ iff $\bigwedge_{i=1}^m (u_i = v_i)$. The empty tuple $()$ is identified with the empty set \emptyset .

By $A(X, Y)$, where X and Y are disjoint tuples of distinct variables, we denote an atom A such that $vars(A) = X \cup Y$. Given the atom $A(X, Y)$, if Y is the k -tuple (Y_1, \dots, Y_k) of distinct variables, and Z is the k -tuple (Z_1, \dots, Z_k) of distinct variables not occurring in X , then by $A(X, Z)$ we denote the atom obtained by replacing in $A(X, Y)$ the variable Y_i by Z_i , for $i = 1, \dots, k$. The

atom $A(X, Y)$ is said to be *functional from the input variables X to the output variables Y with respect to the set P of definite clauses* if

$$(Funct) \quad M(P) \models \forall X, Y, Z. A(X, Y) \wedge A(X, Z) \rightarrow Y = Z$$

The atom $A(X, Y)$ is said to be *total from the input variables X to the output variables Y with respect to the set P of definite clauses* if

$$(Total) \quad M(P) \models \forall X \exists Y. A(X, Y)$$

If $A(X, Y)$ is a total, functional atom from X to Y , we will also write $A(X; Y)$. For instance, with respect to the definite clauses 2–7 shown in Section 2, we have that: (i) `append(xs, ys, zs)` is a total, functional atom from the pair $(\mathbf{xs}, \mathbf{ys})$ of input variables to the output variable \mathbf{zs} , and (ii) `reverse(zs, rs)` is a total, functional atom from the input variable \mathbf{zs} to the output variable \mathbf{rs} .

When referring to the notions of functionality and totality, we will feel free not to mention the set P of definite clauses, if it is understood from the context.

Note that, in our application to program verification, the initial set of clauses is obtained by translating a terminating functional program into CHCs, and hence the functionality Property (*Funct*) and the totality Property (*Total*) hold by construction for any atom $A(X, Y)$ that translates a function defined by that program.

We can extend the functionality and totality notions from atoms to conjunctions of atoms as follows. Let $F(X, Y)$ denote a conjunction A_1, \dots, A_n of n (≥ 1) atoms, where X and Y are disjoint tuples of distinct variables such that $\text{vars}(\{A_1, \dots, A_n\}) = X \cup Y$. Then, $F(X, Y)$ is said to be functional from X to Y if Property (*Funct*) holds for $F(X, Y)$ and $F(X, Z)$, instead of $A(X, Y)$ and $A(X, Z)$, respectively. Similarly, $F(X, Y)$ is said to be total from X to Y if Property (*Total*) holds for $F(X, Y)$, instead of $A(X, Y)$. If $F(X, Y)$ is a total, functional conjunction from X to Y , we will also write $F(X; Y)$.

Now, let us consider a conjunction F of n (≥ 1) total, functional atoms, which (modulo reordering) is equal to ' $A_1(X_1; Y_1), \dots, A_n(X_n; Y_n)$ ', where: (1) the output (tuples of) variables Y_i 's are pairwise disjoint, and (2) for $i = 1, \dots, n$, $(\bigcup_{j=1}^i X_j) \cap Y_i = \emptyset$. Then, F is a total, functional conjunction from X to Y , where $Y = \bigcup_{i=1}^n Y_i$ and $X = (\bigcup_{i=1}^n X_i) \setminus Y$, and hence it can be denoted by $F(X; Y)$. For instance, the conjunction '`append(xs, ys, zs)`, `reverse(zs, rs)`', whose predicates are defined in Section 2, is a total, functional conjunction from the pair $(\mathbf{xs}, \mathbf{ys})$ of input variables to the pair $(\mathbf{zs}, \mathbf{rs})$ of output variables.

4 Transformation Rules for Constrained Horn Clauses

In this section we present the rules that we propose for transforming CHCs, and in particular, for introducing difference predicates, and we prove the soundness of those rules.

4.1 The transformation rules

First, we introduce the following notion of a *stratification* for a set of clauses. Let \mathbb{N} denote the set of the natural numbers. A *level mapping* is a function ℓ :

$Pred \rightarrow \mathbb{N}$. For every predicate p , the natural number $\ell(p)$ is said to be the *level* of p . Level mappings are extended to atoms by stating that the level $\ell(A)$ of an atom A is the level of its predicate symbol. A clause $H \leftarrow c, A_1, \dots, A_n$ is *stratified with respect to the level mapping ℓ* if, for $i = 1, \dots, n$, $\ell(H) \geq \ell(A_i)$. A set P of CHCs is *stratified with respect to ℓ* if all clauses of P are stratified with respect to ℓ . Clearly, for every set P of CHCs, there exists a level mapping ℓ such that P is stratified with respect to ℓ [40].

A *transformation sequence from P_0 to P_n* is a sequence $P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_n$ of sets of CHCs such that, for $i = 0, \dots, n-1$, P_{i+1} is derived from P_i , denoted $P_i \Rightarrow P_{i+1}$, by applying one of the following rules R1–R7. We assume that the initial set P_0 is stratified with respect to a given level mapping ℓ .

The Definition Rule allows us to introduce new predicate definitions.

(R1) Definition Rule. Let D be the clause $newp(X_1, \dots, X_k) \leftarrow c, A_1, \dots, A_m$, where: (1) $newp$ is a predicate symbol in $Pred$ not occurring in the sequence $P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_i$ constructed so far, (2) c is a constraint, (3) the predicate symbols of A_1, \dots, A_m occur in P_0 , and (4) $(X_1, \dots, X_k) \subseteq vars(\{c, A_1, \dots, A_m\})$.

Then, by *definition* we get $P_{i+1} = P_i \cup \{D\}$. We define the level mapping ℓ of $newp$ to be equal to $\max\{\ell(A_i) \mid i = 1, \dots, m\}$.

For $j = 0, \dots, n$, by $Defs_j$ we denote the set of clauses, called *definitions*, introduced by rule R1 during the construction of the prefix $P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_j$ of the transformation sequence $P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_n$. Thus, $Defs_0 = \emptyset$, and for $j = 0, \dots, n$, $Defs_j \subseteq Defs_{j+1}$. Note that, by using rules R2–R7, one may replace a definition occurring in P_h , for some $0 < h < n$, and hence it may happen that $Defs_k \not\subseteq P_k$, for some k such that $h < k \leq n$.

The Unfolding Rule consists in performing a symbolic computation step.

(R2) Unfolding Rule. Let $C: H \leftarrow c, G_L, A, G_R$ be a clause in P_i , where A is an atom. Without loss of generality, we assume that $vars(C) \cap vars(P_0) = \emptyset$. Let $Cls: \{K_1 \leftarrow c_1, B_1, \dots, K_m \leftarrow c_m, B_m\}$, with $m \geq 0$, be the set of clauses in P_0 , such that: for $j = 1, \dots, m$, (1) there exists a most general unifier ϑ_j of A and K_j , and (2) the conjunction of constraints $(c, c_j)\vartheta_j$ is satisfiable. Let $Unf(C, A, P_0)$ be the set $\{(H \leftarrow c, c_j, G_L, B_j, G_R)\vartheta_j \mid j = 1, \dots, m\}$ of clauses.

Then, by *unfolding C with respect to A* , we derive the set $Unf(C, A, P_0)$ and we get $P_{i+1} = (P_i \setminus \{C\}) \cup Unf(C, A, P_0)$.

When we apply rule R2, we say that, for $j = 1, \dots, m$, the atoms in the conjunction $B_j\vartheta_j$ are *derived* from A , and the atoms in the conjunction $(G_L, G_R)\vartheta_j$ are *inherited* from the corresponding atoms in the body of C .

The Folding Rule is a special case of an inverse of the Unfolding Rule.

(R3) Folding Rule. Let $C: H \leftarrow c, G_L, Q, G_R$ be a clause in P_i , and let $D: K \leftarrow d, B$ be a variant of a clause in $Defs_i$. Suppose that: (1) either H is *false* or $\ell(H) \geq \ell(K)$, and (2) there exists a substitution ϑ such that $Q = B\vartheta$ and $\mathbb{D} \models \forall(c \rightarrow d\vartheta)$.

Then, by *folding C using definition D*, we derive clause $E: H \leftarrow c, G_L, K\vartheta, G_R$, and we get $P_{i+1} = (P_i \setminus \{C\}) \cup \{E\}$.

The Clause Deletion Rule removes a clause with an unsatisfiable constraint in its body.

(R4) Clause Deletion Rule. Let $C: H \leftarrow c, G$ be a clause in P_i such that the constraint c is unsatisfiable.

Then, by *clause deletion* we get $P_{i+1} = P_i \setminus \{C\}$.

The Functionality Rule rewrites a functional conjunction of atoms by using Property (*Funct*).

(R5) Functionality Rule. Let $C: H \leftarrow c, G_L, F(X, Y), F(X, Z), G_R$ be a clause in P_i , where $F(X, Y)$ is a functional conjunction of atoms from X to Y with respect to $Definite(P_0) \cup Defs_i$.

Then, by *functionality*, from C we derive $D: H \leftarrow c, Y = Z, G_L, F(X, Y), G_R$, and we get $P_{i+1} = (P_i \setminus \{C\}) \cup \{D\}$.

The Totality Rule rewrites a functional conjunction of atoms by using Property (*Total*).

(R6) Totality Rule. Let $C: H \leftarrow c, G_L, F(X, Y), G_R$ be a clause in P_i such that $Y \cap vars(H \leftarrow c, G_L, G_R) = \emptyset$ and $F(X, Y)$ is a total conjunction of atoms from X to Y with respect to $Definite(P_0) \cup Defs_i$.

Then, by *totality*, from C we derive clause $D: H \leftarrow c, G_L, G_R$, and we get $P_{i+1} = (P_i \setminus \{C\}) \cup \{D\}$.

As mentioned above, the functionality and totality properties hold by construction, and we do not need to prove them when applying rules R5 and R6.

The Differential Replacement Rule replaces a conjunction of atoms by a new conjunction together with an atom defining a relation among the variables of those conjunctions.

(R7) Differential Replacement Rule. Let $C: H \leftarrow c, G_L, F(X; Y), G_R$ be a clause in P_i , and let $D: diff(Z) \leftarrow d, F(X; Y), R(V; W)$ be a variant of a definition clause in $Defs_i$, such that: (1) $F(X; Y)$ and $R(V; W)$ are total, functional conjunctions with respect to $Definite(P_0) \cup Defs_i$, (2) $W \cap vars(C) = \emptyset$, (3) $\mathbb{D} \models \forall(c \rightarrow d)$, and (4) $\ell(H) > \ell(diff(Z))$.

Then, by *differential replacement*, we derive clause $E: H \leftarrow c, G_L, R(V; W), diff(Z), G_R$, and we get $P_{i+1} = (P_i \setminus \{C\}) \cup \{E\}$.

Note that in rule R7 no assumption is made on the set Z of variables, apart from the one deriving from the fact that D is a definition, that is, $Z \subseteq vars(d) \cup X \cup Y \cup V \cup W$.

The transformation algorithm \mathcal{R} for the removal of ADTs, which we will present in Section 5, applies a specific instance of rule R7 (see, in particular, the Diff-Introduce step). The general form of rule R7 that we have now considered, makes it easier to prove the Soundness and Completeness Theorems (see Theorems 1 and 6) we will present below.

4.2 Soundness of the transformation rules

Now we will extend to rules R1–R7 some correctness results that have been proved for the transformation of (constraint) logic programs [23, 24, 47, 52].

Theorem 1 (Soundness of the Transformation Rules). *Let $P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_n$ be a transformation sequence using rules R1–R7. Suppose that the following condition holds:*

(U) *for $i=1, \dots, n-1$, if $P_i \Rightarrow P_{i+1}$ by folding a clause in P_i using a definition $D : H \leftarrow c, B$ in Defs_i , then, for some $j \in \{1, \dots, i-1, i+1, \dots, n-1\}$, $P_j \Rightarrow P_{j+1}$ by unfolding D with respect to an atom A such that $\ell(H) = \ell(A)$. If P_n is satisfiable, then P_0 is satisfiable.*

Thus, to prove the satisfiability of a set P_0 of clauses, it suffices to: (i) construct a transformation sequence $P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_n$, and then (ii) prove that P_n is satisfiable.

The need for Condition (U) in Theorem 1 can be shown by the following example.

Example 1. Let us consider the following initial set of clauses:

P_0 : 1. `false :- p.`
2. `p.`

By rule R1 we introduce the definition:

3. `newp :- p.`

and we get the set $P_1 = \{1,2,3\}$ of clauses. Then, by folding clause 1 using definition 3, we get:

P_2 : 1f. `false :- newp.`
2. `p.`
3. `newp :- p.`

Again, by folding definition 3 using the same definition 3, we get:

P_3 : 1f. `false :- newp.`
2. `p.`
3f. `newp :- newp.`

Now we have that P_3 is satisfiable (being $\{p\}$ its least \mathbb{D} -model), while P_0 is unsatisfiable. This fact is consistent with Theorem 1. Indeed, the transformation sequence $P_0 \Rightarrow P_1 \Rightarrow P_2 \Rightarrow P_3$ does not comply with Condition (U) because during that sequence, definition 3 has not been unfolded. \square

The following example shows that for the application of rule R7, Condition (4) cannot be dropped because, otherwise, Theorem 1 does not hold.

Example 2. Let us consider the initial set of clauses:

P_0 : 1. `false :- r(X,Y).`
2. `r(X,Y) :- f(X,Y).`
3. `f(X,Y) :- Y=0.`

where `f` and `r` are predicates whose arguments are in the set \mathbb{Z} of the integers. Let us assume that the level mapping ℓ is defined as follows: $\ell(\mathbf{f}) = 1$ and $\ell(\mathbf{r}) = 2$. Now, we apply rule R1 and we introduce a new predicate `diff`, and we get:

- $$P_1: \begin{array}{l} 1. \text{ false} \quad :- \text{ r}(X, Y). \\ 2. \text{ r}(X, Y) \quad :- \text{ f}(X, Y). \\ 3. \text{ f}(X, Y) \quad :- Y=0. \\ 4. \text{ diff}(X, W, Y) \quad :- \text{ f}(X, Y), \text{ r}(X, W). \end{array}$$

where, complying with rule R1, we set $\ell(\text{diff})=2$. By applying rule R7, even if Condition (4) is not satisfied, we get:

- $$P_2: \begin{array}{l} 1. \text{ false} \quad :- \text{ r}(X, Y). \\ 2r. \text{ r}(X, Y) \quad :- \text{ r}(X, W), \text{ diff}(X, W, Y). \\ 3. \text{ f}(X, Y) \quad :- Y=0. \\ 4. \text{ diff}(X, W, Y) \quad :- \text{ f}(X, Y), \text{ r}(X, W). \end{array}$$

Now, contrary to the conclusion of Theorem 1, we have that P_0 is unsatisfiable and P_2 is satisfiable, being $\{\text{f}(\mathbf{n}, 0) \mid \mathbf{n} \in \mathbb{Z}\}$ its least \mathbb{Z} -model. Note that the other Conditions (1), (2), and (3) for applying rule R7 do hold. In particular, the atoms $\text{f}(X, Y)$ and $\text{r}(X, Y)$ are total, functional atoms from X to Y with respect to P_0 . \square

The rest of this section is devoted to the proof of Theorem 1.

First, we recall and recast in our framework some definitions and facts taken from the literature [23, 51, 52]. Besides the rules presented in Section 4.1 above, let us also consider the following rule R8 which, given a transformation sequence $P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_i$, for some $i \geq 0$, allows us to extend it by constructing a new set P_{i+1} of CHCs such as $P_i \Rightarrow P_{i+1}$.

(R8) Goal Replacement Rule. Let $C: H \leftarrow c, c_1, G_L, G_1, G_R$ be a clause in P_i . If in clause C we replace c_1, G_1 by c_2, G_2 , we derive clause $D: H \leftarrow c, c_2, G_L, G_2, G_R$, and we get $P_{i+1} = (P_i \setminus \{C\}) \cup \{D\}$.

Now let us introduce two particular kinds of Goal Replacement Rule: (i) *body weakening*, and (ii) *body strengthening*. First, for the Goal Replacement Rule, we need to consider the following two tuples of variables:

$$\begin{aligned} T_1 &= \text{vars}(\{c_1, G_1\}) \setminus \text{vars}(\{H, c, G_L, G_R\}), \quad \text{and} \\ T_2 &= \text{vars}(\{c_2, G_2\}) \setminus \text{vars}(\{H, c, G_L, G_R\}). \end{aligned}$$

The Goal Replacement Rule is said to be a *body weakening* if the following two conditions hold:

- $$\begin{aligned} \text{(W.1)} \quad & M(\text{Definite}(P_0) \cup \text{Defs}_i) \models \forall (c_1 \wedge G_1 \rightarrow \exists T_2. c_2 \wedge G_2) \\ \text{(W.2)} \quad & \ell(H) > \ell(A), \text{ for every atom } A \text{ occurring in } G_2 \text{ and not in } G_1. \end{aligned}$$

The Goal Replacement Rule is said to be a *body strengthening* if the following condition holds:

- $$\text{(S)} \quad M(\text{Definite}(P_0) \cup \text{Defs}_i) \models \forall (c_2 \wedge G_2 \rightarrow \exists T_1. c_1 \wedge G_1).$$

Usually, in the literature the Goal Replacement Rule is presented by considering the conjunction of Conditions (W.1) and (S), thereby considering the quantified equivalence $\forall ((\exists T_1. c_1 \wedge G_1) \leftrightarrow (\exists T_2. c_2 \wedge G_2))$. We have split that equivalence into the two associated implications. This has been done because, when proving the Soundness result (see Theorem 1 in this section) and the Completeness result (see Theorem 6 in Section 6.1), it is convenient to present the

preservation of the least \mathbb{D} -models into two parts as specified by the following two theorems.

Theorem 2. *Let D_0, \dots, D_n be sets of definite CHCs and let $D_0 \Rightarrow \dots \Rightarrow D_n$ be a transformation sequence constructed using rules R1 (Definition), R2 (Unfolding), R3 (Folding), and R8 (Goal Replacement). Suppose that Condition (U) of Theorem 1 holds and all goal replacements are body weakenings. Then $M(D_0 \cup \text{Defs}_n) \subseteq M(D_n)$.*

Theorem 3. *Let D_0, \dots, D_n be sets of definite CHCs and let $D_0 \Rightarrow \dots \Rightarrow D_n$ be a transformation sequence constructed using rules R1 (Definition), R2 (Unfolding), R3 (Folding), and R8 (Goal Replacement). Suppose that, for all applications of R3, Condition (E) holds (see Definition 9 in Section 6) and all goal replacements are body strengthenings. Then $M(D_0 \cup \text{Defs}_n) \supseteq M(D_n)$.*

For the proof of Theorems 2 and 3 we refer to the results presented in the literature [23, 51, 52]. The correctness of the transformation rules with respect to the least Herbrand model semantics has been first proved in the landmark paper by Tamaki and Sato [51]. In a subsequent technical report [52], the same authors extended that result by introducing the notion of the *level* of an atom, which we also use in this paper (see the notion defined at the beginning of Section 4.1).

The use of atom levels allows less restrictive applicability conditions on the Folding and Goal Replacement Rules. Later, Etalle and Gabbrielli [23] extended Tamaki and Sato's results to the \mathbb{D} -model semantics of *constraint logic programs* (in the terminology used in this paper, a constraint logic program is a set of definite constrained Horn clauses).

There are three main differences between our presentation of the correctness results for the transformation rules with respect to the presentation considered in the literature [23, 51, 52].

First, as already mentioned, we kept the two Conditions (E) and (S), which guarantee the inclusion $M(D_0 \cup \text{Defs}_n) \supseteq M(D_n)$ (called *Partial Correctness* by Tamaki and Sato [52]), separated from the three Conditions (U), (W.1), and (W.2), which guarantee the reverse inclusion $M(D_0 \cup \text{Defs}_n) \subseteq M(D_n)$. All five conditions together guarantee the equality $M(D_0 \cup \text{Defs}_n) = M(D_n)$ (called *Total Correctness* by Tamaki and Sato [52]).

Second, Tamaki and Sato's conditions for the correctness of the Goal Replacement Rule are actually more general than ours, as they use a well-founded relation which is based on atom levels and also on a suitable measure (called *weight-tuple measure* [52]) of the successful derivations of an atom in $M(D_0 \cup \text{Defs}_n)$. Our simpler conditions straightforwardly imply Tamaki and Sato's ones, and are sufficient for our purposes in the present paper.

Third, Tamaki and Sato papers [51, 52] do not consider constraints, whereas Etalle and Gabbrielli results for constraint logic programs do not consider Goal Replacement [23]. However, Tamaki and Sato's proofs can easily be extended to constraint logic programs by simply dealing with atomic constraints as atoms with level 0 and assigning positive levels to all other atoms.

From Theorem 2, we get the following Theorem 4, which relates the satisfiability of sets of clauses obtained by applying the transformation rules to the satisfiability of the original sets of clauses.

Theorem 4. *Let $P_0 \Rightarrow \dots \Rightarrow P_n$ be a transformation sequence constructed using rules R1 (Definition), R2 (Unfolding), R3 (Folding), and R8 (Goal Replacement). Suppose that Condition (U) of Theorem 1 holds and all goal replacements are body weakenings. If P_n is satisfiable, then P_0 is satisfiable.*

Proof. First, we observe that P_0 is satisfiable iff $P_0 \cup \text{Defs}_n$ is satisfiable. Indeed, we have that: (i) if \mathcal{M} is a \mathbb{D} -model of P_0 , then the \mathbb{D} -interpretation $\mathcal{M} \cup \{\text{newp}(a_1, \dots, a_k) \mid \text{newp} \text{ is a head predicate in } \text{Defs}_n \text{ and } a_1, \dots, a_k \text{ are ground terms}\}$ is a \mathbb{D} -model of $P_0 \cup \text{Defs}_n$, and (ii) if \mathcal{M} is a \mathbb{D} -model of $P_0 \cup \text{Defs}_n$, then all clauses of P_0 are true in \mathcal{M} , and hence \mathcal{M} is a \mathbb{D} -model of P_0 .

Then, let us consider a new transformation sequence $P'_0 \Rightarrow \dots \Rightarrow P'_n$ obtained from the sequence $P_0 \Rightarrow \dots \Rightarrow P_n$ by replacing each occurrence of *false* in the head of a clause by a fresh, new predicate symbol, say f . P'_0, \dots, P'_n are sets of definite clauses, and thus, for $i = 0, \dots, n$, $\text{Definite}(P'_i) = P'_i$. The sequence $P'_0 \Rightarrow \dots \Rightarrow P'_n$ satisfies the hypotheses of Theorem 2, and hence $M(P'_0 \cup \text{Defs}_n) \subseteq M(P'_n)$. We have that:

P_n is satisfiable
implies $P'_n \cup \{\neg f\}$ is satisfiable
implies $f \notin M(P'_n)$
implies, by Theorem 2, $f \notin M(P'_0 \cup \text{Defs}_n)$
implies $P'_0 \cup \text{Defs}_n \cup \{\neg f\}$ is satisfiable
implies $P_0 \cup \text{Defs}_n$ is satisfiable
implies P_0 is satisfiable. □

Now, in order to prove Theorem 1 of Section 4, which states the soundness of rules R1–R7, we show that rules R4–R7 are all body weakenings.

An application of rule R4 (Clause Deletion), by which we delete clause C : $H \leftarrow c, G$, whenever the constraint c is unsatisfiable, is equivalent to the replacement of the body of clause C by *false*. Since c is unsatisfiable, we have that:

$$M(\text{Definite}(P_0) \cup \text{Defs}_i) \models \forall (c \wedge G \rightarrow \text{false})$$

and Condition (W.1) of rule R8 holds. Also Condition (W.2), that is:

$$\ell(H) > \ell(A), \text{ for every atom } A \text{ occurring in } \text{false}$$

trivially holds, because there are no atoms in *false*. Thus, the replacement of the body of clause $H \leftarrow c, G$ by *false* is a body weakening.

Let us now consider rule R5 (Functionality). Let $F(X, Y)$ be a conjunction of atoms that defines a functional relation from X to Y , that is, Property (*Funct*) of Section 3 holds for $F(X, Y)$. When rule R5 is applied whereby a conjunction

$F(X, Y), F(X, Z)$ is replaced by the new conjunction $Y = Z, F(X, Y)$, we have that:

$$M(\text{Definite}(P_0) \cup \text{Defs}_i) \models \forall(F(X, Y) \wedge F(X, Z) \rightarrow Y = Z)$$

and hence Condition (W.1) of rule R8 holds. When this replacement is performed, also Condition (W.2) trivially holds, and thus rule R5 is a body weakening.

An application of rule R6 (Totality) replaces a conjunction $F(X, Y)$ by *true* (that is, the empty conjunction), which is implied by any formula. Hence Conditions (W.1) and (W.2) trivially hold, and rule R6 is a body weakening.

For rule R7 (Differential Replacement) we prove the following lemma. Recall that by $F(X; Y)$ we denote a conjunction of atoms that defines a total, functional relation from X to Y .

Lemma 1. *Let us consider a transformation sequence $P_0 \Rightarrow \dots \Rightarrow P_i$ and a clause $C: H \leftarrow c, G_L, F(X; Y), G_R$ in P_i . Let us assume that by applying rule R7 on clause C using the definition clause*

$$D: \text{diff}(Z) \leftarrow d, F(X; Y), R(V; W),$$

where: (D1) $W \cap \text{vars}(C) = \emptyset$, and (D2) $\mathbb{D} \models \forall(c \rightarrow d)$, we derive clause

$$E: H \leftarrow c, G_L, R(V; W), \text{diff}(Z), G_R$$

and we get the new set $P_{i+1} = (P_i \setminus \{C\}) \cup \{E\}$ of clauses. Then,

$$M(\text{Definite}(P_0) \cup \text{Defs}_i) \models \forall(c \wedge F(X; Y) \rightarrow \exists W. (R(V; W) \wedge \text{diff}(Z))).$$

Proof. Let \mathcal{M} denote $M(\text{Definite}(P_0) \cup \text{Defs}_i)$. Since, $R(V; W)$ is a total, functional conjunction from V to W with respect to $\text{Definite}(P_0) \cup \text{Defs}_i$, we have:

$$\mathcal{M} \models \forall(c \wedge F(X; Y) \rightarrow \exists W. R(V; W)) \quad (\alpha)$$

Since, by Condition (D1), none of the variables in W occurs in C , from definition D , we get:

$$\mathcal{M} \models \forall(d \wedge F(X; Y) \wedge R(V; W) \rightarrow \text{diff}(Z)) \quad (\beta)$$

From (α) , (β) , and Condition (D2), we get the thesis. \square

From Lemma 1 it follows that rule R7, which replaces in the body of clause $C: H \leftarrow c, G_L, F(X; Y), G_R$ the conjunction $F(X; Y)$ by the new conjunction $R(V; W), \text{diff}(Z)$, is a body weakening, assuming that $\ell(H) > \ell(\text{diff}(Z))$. Recall that, since clause D is a definition clause, we have that $\ell(\text{diff}(Z)) \geq \ell(R)$, and thus we have that $\ell(H) > \ell(R)$.

The following lemma summarizes the facts we have shown above about rules R4–R7.

Lemma 2. *The applications of rules R4–R7 are all body weakenings.*

Finally, having proved Lemma 2, we can present the proof of Theorem 1.

Proof of Theorem 1. Let $P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_n$ be a transformation sequence constructed using rules R1–R7. Then, by Lemma 2, that sequence can also be

constructed by applications of rules R1–R3 together with applications of rule R8 which are all body weakenings. Since by hypothesis of Theorem 1, Condition (U) does hold, by applying Theorem 4 we get the thesis. \square

5 An Algorithm for ADT Removal

In this section we present Algorithm \mathcal{R} for eliminating ADT terms from CHCs by using the transformation rules R1–R7 presented in Section 4 and automatically introducing suitable difference predicates. Then we show that, if Algorithm \mathcal{R} terminates, it transforms a set Cls of clauses into a new set $TransfCls$ where every argument of every predicate has a basic type. Theorem 1 (see Section 4.2) guarantees that if $TransfCls$ is satisfiable, then also Cls is satisfiable.

5.1 The ADT removal Algorithm \mathcal{R}

Algorithm \mathcal{R} (see Figure 1) removes ADT terms starting from the set Gs of goals in Cls . Initially, those goals are all collected in the set $InCls$. The set $Defs$ collects the definitions of the new predicates introduced by applications of rule R1 during the execution of Algorithm \mathcal{R} . Initially, we have that $Defs = \emptyset$.

Algorithm \mathcal{R}

Input: A set Cls of clauses and a level mapping ℓ of the predicates occurring in Cls .

Output: A set $TransfCls$ of clauses that have basic types.

Let $Cls = Ds \cup Gs$, where Ds is a set of definite clauses and Gs is a set of goals;

$InCls := Gs$; $Defs := \emptyset$; $TransfCls := \emptyset$;

while $InCls \neq \emptyset$ **do**

- $Diff\text{-}Define\text{-}Fold(InCls, Defs, NewDefs, FldCls)$;
 - $Unfold(NewDefs, Ds, UnfCls)$;
 - $Replace(UnfCls, Ds, RCls)$;
- $InCls := RCls$; $Defs := Defs \cup NewDefs$; $TransfCls := TransfCls \cup FldCls$;
-

Fig. 1. The ADT removal algorithm \mathcal{R} .

Algorithm \mathcal{R} iterates a sequence made out of the following three procedures.

(1) *Procedure Diff-Define-Fold* introduces, by rule R1, a set $NewDefs$ of suitable new predicate definitions. By applications of the Folding Rule R3 and, possibly, of the Differential Replacement Rule R7, using clauses in $Defs \cup NewDefs$, the procedure removes the ADT terms from the input set $InCls$ of clauses.

The bodies (but not the heads) of the clauses in $NewDefs$ contain ADT terms, and thus they need to be transformed to remove those terms.

(2) *Procedure Unfold* performs some steps of symbolic evaluation of the newly introduced definitions by applying the Unfolding Rule R2 to the clauses occurring in $NewDefs$.

(3) *Procedure Replace* removes clauses that have an unsatisfiable body by applying rule R4, and also exploits the functionality and totality properties of the predicates by applying rules R5 and R6, respectively.

The clauses with ADTs obtained after the *Replace* procedure and the new predicate definitions introduced at each iteration, are added to *InCls* and *Defs*, respectively. Algorithm \mathcal{R} terminates when the set *InCls* of clauses becomes empty because no new definitions need to be introduced to perform folding steps.

Note that Algorithm \mathcal{R} takes as input also a level mapping ℓ for the predicates occurring in *Cls*. In our implementation, however, no function ℓ is actually provided and, instead, a suitable level mapping is constructed during the execution of the algorithm itself. We do this construction by following a general constraint-based approach for guaranteeing the correctness of logic program transformations [43]. In particular, given an initially empty set L of constraints, each time \mathcal{R} applies a transformation rule whose soundness depends on the satisfaction of a constraint on the predicate levels (see, in particular the conditions in rules R1, R3, R7, and Condition (U) in Theorem 1 for R2), that constraint is added to the set L . For the soundness of Algorithm \mathcal{R} , it is required that at the end of its execution, the set L be satisfiable. A solution of L provides the level mapping ℓ to be constructed. In order not to burden the presentation with too many technical details, we will not present here the actual constraint handling mechanism used for the construction of the function ℓ .

Example 3 (Reverse). Throughout this section we will use the *Reverse* example of Section 2 as a running example for illustrating an application of the ADT removal algorithm \mathcal{R} . In that example, the set *Cls* of clauses given as input to \mathcal{R} consists of clauses 1–9, with $Gs = \{1\}$ and $Ds = \{2, \dots, 9\}$. Thus, *InCls* is initialized to $\{1\}$. We assume that the following level mapping ℓ is associated with the predicates occurring in clauses 1–9: $\ell(\text{append}) = \ell(\text{reverse}) = 2$, and $\ell(\text{snoc}) = \ell(\text{len}) = 1$. \square

5.2 Procedure *Diff-Define-Fold*

In order to present the *Diff-Define-Fold* procedure used by Algorithm \mathcal{R} , first we introduce the following notions.

Given a conjunction G of atoms, by $bvars(G)$ we denote the set of variables in G that have a basic type. Similarly, by $adt-vars(G)$ we denote the set of variables in G that have an ADT type.

Definition 1. *We say that an atom (or a clause) has basic types if all its arguments (or atoms, respectively) have a basic type. An atom (or a clause) has ADTs if at least one of its arguments (or atoms, respectively) has an ADT type.*

Definition 2. *Given a set (or a conjunction) S of atoms, $SharingBlocks(S)$ denotes the partition of S with respect to the reflexive, transitive closure, denoted \Downarrow_S , of the relation \downarrow_S defined as follows. Given two atoms A_1 and A_2 in S , $A_1 \downarrow_S A_2$ holds iff $adt-vars(A_1) \cap adt-vars(A_2) \neq \emptyset$. The elements of the partition are called the sharing blocks of S . We say that S is connected if $SharingBlocks(S) = \{S\}$.*

Definition 3. A generalization of a pair (c_1, c_2) of constraints is a constraint, denoted $\alpha(c_1, c_2)$, such that $\mathbb{D} \models \forall(c_1 \rightarrow \alpha(c_1, c_2))$ and $\mathbb{D} \models \forall(c_2 \rightarrow \alpha(c_1, c_2))$.

In particular, we consider the following generalization operator based on *widening* [10, 25]. Suppose that c_1 is the conjunction (a_1, \dots, a_m) of atomic constraints, then $\alpha(c_1, c_2)$ is defined to be the conjunction of all a_i 's in (a_1, \dots, a_m) such that $\mathbb{D} \models \forall(c_2 \rightarrow a_i)$. In order to improve the efficacy of generalization, when some of the a_i 's are *LIA* equalities, they are split into conjunctions of *LIA* inequalities before applying widening.

Definition 4. For any constraint c and tuple V of variables, the projection of c onto V is a constraint $\pi(c, V)$ such that: (i) $\text{vars}(\pi(c, V)) \subseteq V$, and (ii) $\mathbb{D} \models \forall(c \rightarrow \pi(c, V))$.

In our implementation, $\pi(c, V)$ is computed by applying to the formula $\exists Y. c$, where $Y = \text{vars}(c) \setminus V$, a quantifier elimination algorithm for the theories of booleans and *rational* (not integer) numbers. This implementation is safe in our context, because it guarantees properties (i) and (ii) of Definition 4, and avoids relying on modular arithmetic, as usually done when eliminating quantifiers in *LIA* [45].

Definition 5. For two conjunctions G_1 and G_2 of atoms, we say that G_1 atom-wise subsumes G_2 , denoted $G_1 \preceq G_2$, if G_1 is the conjunction (A_1, \dots, A_n) and there exists a subconjunction (B_1, \dots, B_n) of atoms of G_2 (modulo reordering) and substitutions $(\vartheta_1, \dots, \vartheta_n)$ such that, for $i = 1, \dots, n$, we have that $B_i = A_i \vartheta_i$.

Now let us present the *Diff-Define-Fold* procedure (see Figure 2). At each iteration of the body of the **for** loop, the *Diff-Define-Fold* procedure removes the ADT terms occurring in a sharing block B of the body of a clause $C : H \leftarrow c, B, G'$ of *InCls* (initially, *InCls* is a set of goals whose head is *false*). This is done by possibly introducing some new definitions using the Definition Rule R1 and applying the Folding Rule R3. To allow folding, some applications of the Differential Replacement Rule R7 may be needed. We have the following four cases.

- **(Fold).** In this case the ADT terms in B can be removed by folding using a definition that has already been introduced. In particular, let us suppose that B is an instance, via a substitution ϑ , of the conjunction of atoms in the body of a definition D introduced at a previous iteration of the *Diff-Define-Fold* procedure, and constraint c in C entails the constraint in D . Since we have assumed that all terms of a basic type occurring in an atom are distinct variables (see Section 3), and the variables of D can be freely renamed, we require that ϑ acts on ADT variables only, and hence it is the identity on the variables of a basic type. A similar assumption is also made in the next two cases (Generalize) and (Diff-Introduce). Then, we remove the ADT arguments occurring in B by folding C using D . Indeed, by construction, all variables in the head of every definition introduced by Algorithm \mathcal{R} have a basic type.

Procedure *Diff-Define-Fold*(*InCls*, *Defs*, *NewDefs*, *FldCls*)
Input: A set *InCls* of clauses and a set *Defs* of definitions;
Output: A set *NewDefs* of definitions and a set *FldCls* of clauses with basic types.

NewDefs := \emptyset ; *FldCls* := \emptyset ;
for each clause $C: H \leftarrow c, G$ in *InCls* **do**
 if C has basic types **then** $InCls := InCls \setminus \{C\}$; $FldCls := FldCls \cup \{C\}$
 else
 let C be $H \leftarrow c, B, G'$ where B is a sharing block in G such that B contains at least one atom that has ADTs;
 • **(Fold)** **if** there is a clause $D: newp(U) \leftarrow d, B'$, which is a variant of a clause in $Defs \cup NewDefs$, with $U = bvars(\{d, B'\})$, such that: (i) $B = B'\vartheta$, for some substitution ϑ acting on $adt\text{-}vars(B')$ only, and (ii) $\mathbb{D} \models \forall(c \rightarrow d)$, **then** fold C using D and derive $E: H \leftarrow c, newp(U), G'$;
 • **(Generalize)** **else if** there is a clause $D: newp(U) \leftarrow d, B'$, which is a variant of a clause in $Defs \cup NewDefs$, with $U = bvars(\{d, B'\})$, such that: (i) $B = B'\vartheta$, for some substitution ϑ acting on $adt\text{-}vars(B')$ only, and (ii) $\mathbb{D} \not\models \forall(c \rightarrow d)$, **then**
 then
 introduce definition $GenD: genp(U) \leftarrow \alpha(d, c), B'$
 fold C using $GenD$ and derive $E: H \leftarrow c, genp(U), G'$;
 $NewDefs := NewDefs \cup \{GenD\}$;
 • **(Diff-Introduce)** **else if** there is a clause $D: newp(U) \leftarrow d, B'$, which is a variant of a clause in $Defs \cup NewDefs$, with $U = bvars(\{d, B'\})$ and $B' \preceq B$, **then**
 take a maximal connected subconjunction M of B , if any, such that:
 (i) $B = (M, F(X; Y))$, for some non-empty conjunction $F(X; Y)$, (ii) $B'\vartheta = (M, R(V; W))$, for some substitution ϑ acting on $adt\text{-}vars(B')$ only and $W \cap vars(C) = \emptyset$, and (iii) for every atom A in $F(X; Y)$, $\ell(H) > \ell(A)$;
 introduce definition $\hat{D}: diff(Z) \leftarrow \pi(c, X), F(X; Y), R(V; W)$
 where $Z = bvars(\{F(X; Y), R(V; W)\})$;
 $NewDefs := NewDefs \cup \{\hat{D}\}$;
 replace $F(X; Y)$ by $(R(V; W), diff(Z))$ in C , and derive clause
 $C': H \leftarrow c, M, R(V; W), diff(Z), G'$;
 if $\mathbb{D} \models \forall(c \rightarrow d)$
 then fold C' using D and derive $E: H \leftarrow c, newp(U), diff(Z), G'$;
 else introduce definition $GenD: genp(U) \leftarrow \alpha(d, c), B'$;
 fold C' using $GenD$ and derive $E: H \leftarrow c, genp(U), diff(Z), G'$;
 $NewDefs := NewDefs \cup \{GenD\}$;
 • **(Project)** **else**
 introduce definition $ProjC: newp(U) \leftarrow \pi(c, Z), B$ where $U = bvars(B)$
 and Z are the input variables of a basic type in B ;
 fold C using $ProjC$ and derive clause $E: H \leftarrow c, newp(U), G'$;
 $NewDefs := NewDefs \cup \{ProjC\}$;
 $InCls := (InCls \setminus \{C\}) \cup \{E\}$;

Fig. 2. The *Diff-Define-Fold* procedure. According to rule R1, the level of every new predicate (either *genp*, or *diff*, or *newp*) introduced by the procedure, is equal to the maximum level of the atoms occurring in the body of its definition.

- **(Generalize)**. Suppose that the previous case does not apply. Suppose also that there exists a definition D , introduced at a previous iteration of the *Diff-Define-Fold* procedure, such that the sharing block B is an instance of the conjunction B' of the atoms in the body of D and, unlike the (Fold) case, the constraint c in C does *not* entail the constraint d in D . We introduce a new definition $GenD: genp(U) \leftarrow \alpha(d, c), B'$, where: (i) by construction, the constraint $\alpha(d, c)$ is a generalization of d such that c entails d , and (ii) U is the tuple of the variables of a basic type in (d, B) . Then, we remove the ADT arguments occurring in B by folding C using $GenD$.

- **(Diff-Introduce)**. Suppose that the previous two cases do not apply because the sharing block B in clause C is not an instance of the conjunction of atoms in the body of any definition introduced at a previous iteration of the procedure. Suppose, however, that B *partially matches* the body of an already introduced definition $D: newp(U) \leftarrow d, B'$, that is, (i) $B = (M, F(X; Y))$, and (ii) for some substitution ϑ acting on *adt-vars*(B') only, $B'\vartheta = (M, R(V; W))$ (see Figure 2 for details). Then, we introduce a difference predicate $diff$ defined by the clause $\hat{D}: diff(Z) \leftarrow \pi(c, X), F(X; Y), R(V; W)$, where $Z = bvars(\{F(X; Y), R(V; W)\})$ and, by rule R7, we replace the conjunction $F(X; Y)$ by the new conjunction $(R(V; W), diff(Z))$ in the body of C , thereby deriving C' . Finally, we remove the ADT arguments in B by folding C' using either D (if c entails d) or a clause $GenD$ whose constraint is the generalization $\alpha(d, c)$ of the constraint d (if c does *not* entail d) (again, see Figure 2 for details).

- **(Project)**. Suppose that none of the previous three cases apply. Then, we first introduce a new definition $ProjC: newp(U) \leftarrow \pi(c, Z), B$, where $U = bvars(B)$ and Z are the input variables of basic types in B , and then we can remove the ADT arguments occurring in the sharing block B by folding C using $ProjC$.

Example 4 (Reverse, Continued). The body of goal 1 (see Section 2) has a single sharing block, that is,

$B_1: \text{append}(Xs, Ys, Zs), \text{reverse}(Zs, Rs), \text{len}(Xs, N0), \text{len}(Ys, N1), \text{len}(Rs, N2)$

Indeed, we have that $\text{append}(Xs, Ys, Zs)$ shares a list variable with each of atoms $\text{reverse}(Zs, Rs)$, $\text{len}(Xs, N0)$, and $\text{len}(Ys, N1)$, and atom $\text{reverse}(Zs, Rs)$ shares a list variable with $\text{len}(Rs, N2)$. None of the first three cases (Fold), (Generalize), or (Diff-Introduce) applies, because $Defs \cup NewDefs$ is the empty set. Thus, Algorithm \mathcal{R} introduces the following new definition (see also Section 2):

D1. $\text{new1}(N0, N1, N2) :- \text{append}(Xs, Ys, Zs), \text{reverse}(Zs, Rs), \text{len}(Xs, N0),$
 $\text{len}(Ys, N1), \text{len}(Rs, N2).$

where: (i) new1 is a new predicate symbol, (ii) the body is the sharing block B_1 , (iii) $N0, N1, N2$ are the variables of basic types in B_1 , and (iv) the constraint is the empty conjunction true , that is, the projection of the constraint $N2 = \setminus N0 + N1$ occurring in goal 1 onto the input variables of basic types in B_1 (i.e., the empty set, as $N0, N1, N2$ are all output variables). In accordance with rule R1, we set $\ell(\text{new1}) = \max\{\ell(\text{append}), \ell(\text{reverse}), \ell(\text{len})\} = 2$.

By folding, from goal 1 we derive a new goal without occurrences of list variables:

10. `false :- N2=\=N0+N1, new1(N0,N1,N2).`

The presentation of this example will continue in Example 5 (see Section 5.3). \square

5.3 Procedures *Unfold* and *Replace*

The *Diff-Define-Fold* procedure may introduce new definitions with ADTs in their bodies (see, for instance, clause D1 defining predicate `new1` in Example 4). These definitions are added to *NewDefs* and transformed by the *Unfold* and *Replace* procedures.

Procedure *Unfold* (see Figure 3) repeatedly applies rule R2 in two phases. In Phase 1 the procedure unfolds a given clause in *NewDefs* with respect to so-called *source* atoms in its body. Recalling that each atom is the relational translation of a function call, the source atoms represent innermost function calls in the functional expression corresponding to the clause body. The unfolding steps of Phase 1 may determine, by unification, the instantiation of some input variables. Then, in Phase 2 these instantiations are taken into account for performing further unfolding steps. Indeed, the procedure selects for unfolding only atoms whose input arguments are instances of the corresponding arguments in the heads of their matching clauses.

Procedure *Unfold*(*NewDefs*, *Ds*, *UnfCls*)

Input: A set *NewDefs* of definitions and a set *Ds* of definite clauses;

Output: A set *UnfCls* of definite clauses.

UnfCls := *NewDefs*;

Phase 1.	<ul style="list-style-type: none"> - For each clause <i>C</i> in <i>UnfCls</i>, mark as unfoldable a set <i>S</i> of atoms in the body of <i>C</i> such that: (i) there is an atom <i>A</i> in <i>S</i> with $\ell(H) = \ell(A)$, where <i>H</i> is the head of <i>C</i>, and (ii) all atoms in $S \setminus \{A\}$ are source atoms such that every source variable of the body of <i>C</i> occurs in <i>S</i>; - while there exists a clause <i>C</i>: $H \leftarrow c, L, A, R$ in <i>UnfCls</i>, for some conjunctions <i>L</i> and <i>R</i> of atoms, such that <i>A</i> is an unfoldable atom do $\underline{UnfCls := (UnfCls \setminus \{C\}) \cup Unf(C, A, Ds);}$
Phase 2.	<ul style="list-style-type: none"> - Mark as unfoldable all atoms in the body of each clause in <i>UnfCls</i>; - while there exists a clause <i>C</i>: $H \leftarrow c, L, A, R$ in <i>UnfCls</i>, for some conjunctions <i>L</i> and <i>R</i> of atoms, such that <i>A</i> is a head-instance with respect to <i>Ds</i> and <i>A</i> is either unfoldable or descending do $\underline{UnfCls := (UnfCls \setminus \{C\}) \cup Unf(C, A, Ds);}$

Fig. 3. The *Unfold* procedure.

In order to present the *Unfold* procedure in a formal way, we need the following notions.

Definition 6. *A variable X occurring in a conjunction G of atoms is said to be a source variable if it is an input variable for an atom in G and not an output variable of any atom in G . An atom A in a conjunction G of atoms is said to be a source atom if all its input variables are source variables.*

For instance, in clause 1 of Section 2, where the input variables of the atoms `append(Xs,Ys,Zs)`, `reverse(Zs,Rs)`, `len(Xs,N0)`, `len(Ys,N1)`, and `len(Rs,N2)` are (Xs, Ys) , Zs , Xs , Ys , and Rs , respectively, there are the following three source atoms: `append(Xs,Ys,Zs)`, `len(Xs,N0)`, and `len(Ys,N1)`. These three atoms correspond to the innermost function calls which occur in the functional expression `len(reverse(append xs ys)) ≠ (len xs) + (len ys)` corresponding to the clause body.

Definition 7. *An atom $A(X;Y)$ in the body of clause $C: H \leftarrow c, L, A(X;Y), R$ is a head-instance with respect to a set Ds of clauses if, for every clause $K \leftarrow d, B$ in Ds such that: (1) there exists a most general unifier ϑ of $A(X;Y)$ and K , and (2) the constraint $(c,d)\vartheta$ is satisfiable, we have that ϑ is a variable renaming for X .*

Thus, $A(X;Y)$ is a head-instance, if for all clause heads K in Ds the input variables X are not instantiated by unification with K . For instance, with respect to the set $\{2, 3\}$ of clauses of Section 2, the atom `append([X|Xs],Ys,Zs)` is a head-instance, while the atom `append(Xs,Ys,Zs)` is not.

Recall that in a set Cls of clauses, predicate p immediately depends on predicate q , if in Cls there is a clause of the form $p(\dots) \leftarrow \dots, q(\dots), \dots$. The *depends on* relation is the transitive closure of the *immediately depends on* relation [2].

Definition 8. *Let \prec be a well-founded ordering on tuples of terms such that, for all tuples of terms t and u , if $t \prec u$, then, for all substitutions ϑ , $t\vartheta \prec u\vartheta$. A predicate p is descending with respect to \prec if, for all clauses, $p(t;u) \leftarrow c, p_1(t_1;u_1), \dots, p_n(t_n;u_n)$, for $i = 1, \dots, n$, if p_i depends on p , then $t_i \prec t$. An atom is descending if its predicate is descending.*

The well-founded ordering \prec we use in our implementation is based on the *subterm* relation and is defined as follows: $(u_1, \dots, u_m) \prec (v_1, \dots, v_n)$ if for every u_i there exists v_j such that u_i is a (non necessarily strict) subterm of v_j , and there exists u_i which is a strict subterm of some v_j . For instance, the predicates `append`, `reverse`, `snoc`, and `len` in our running example are all descending.

To control the application of rule R2 in Phases 1 and 2 of the *Unfold* procedure we mark as *unfoldable* some atoms in the body of a clause. If we unfold with respect to atom A clause $C: H \leftarrow c, L, A, R$, then the marking of the clauses in $Unf(C, A, Ds)$ is done as follows: (i) each atom derived from A is not marked as unfoldable, and (ii) each atom A'' inherited from an atom A' , different from A , in the body of C is marked as unfoldable iff A' is marked as unfoldable.

In Phase 1, for each clause C in *NewDefs* the procedure marks as unfoldable a non-empty set S of atoms in the body of C consisting of: (i) an atom A such that $\ell(H) = \ell(A)$, where H is the head of C , and (ii) a set of source atoms (possibly including A) such that every source variable of the body of C occurs in S . Then, the procedure unfolds with respect to all unfoldable atoms. Note that atom A exists because, by construction, when we introduce a new predicate during the *Diff-Define-Fold* procedure, we set the level of the new predicate to the maximal level of an atom in the body of its definition. The unfolding with

respect to A enforces Condition (U) of Theorem 1, and hence the soundness of Algorithm \mathcal{R} .

In Phase 2 the instantiations of input variables determined by the unfolding steps of Phase 1 are taken into account for further applications of rule R2. Indeed, clauses are unfolded with respect to atoms which are head-instances and, in particular, unfolding with respect to head-instances which are descending atoms, is repeated until no such atoms are present.

The termination of the procedure *Unfold* is ensured by the following two facts: (i) if a clause C has $n (\geq 1)$ atoms marked as unfoldable, and clause C is unfolded with respect to an atom A that is marked as unfoldable, then each clause in $Unf(C, A, Ds)$ has $n-1$ atoms marked as unfoldable, and (ii) since \prec is a well-founded ordering, it is not possible to perform an infinite sequence of applications of the Unfolding Rule R2 with respect to descending atoms.

Example 5 (Reverse, Continued). The *Unfold* procedure marks as unfoldable atom $\text{append}(Xs, Ys, Zs)$ in the body of clause D1, which has the same level as the head of the clause. Atom $\text{append}(Xs, Ys, Zs)$ is also a source atom containing all the input variables of the body of clause D1 (that is, Xs and Ys). Then, by unfolding clause D1 with respect to $\text{append}(Xs, Ys, Zs)$, we get:

```
11. new1(N0,N1,N2) :- reverse(Ys,Rs), len([],N0), len(Ys,N1), len(Rs,N2).
12. new1(N0,N1,N2) :- append(Xs,Ys,Zs), reverse([X|Zs],Rs), len([X|Xs],N0),
    len(Ys,N1), len(Rs,N2).
```

Now, atoms $\text{len}([],N0)$, $\text{reverse}([X|Zs],Rs)$, and $\text{len}([X|Xs],N0)$ are all head-instances, and hence the procedure unfolds clauses 11 and 12 with respect to these atoms. We get:

```
13. new1(N0,N1,N2) :- N0=0, reverse(Zs,Rs), len(Zs,N1), len(Rs,N2).
14. new1(N01,N1,N21) :- N01=N0+1, append(Xs,Ys,Zs), reverse(Zs,Rs),
    len(Xs,N0), len(Ys,N1), snoc(Rs,X,R1s),
    len(R1s,N21).
```

The presentation of this transformation will continue in Example 6 below. \square

Procedure *Replace* (see Figure 4) applies rules R4, R5, and R6 as long as possible. *Replace* terminates because each application of one of those rules decreases the number of atoms.

Example 6 (Reverse, Continued). Neither rule R5 nor rule R6 is applicable to clauses 13 and 14. Thus, the first iteration of the body of the **while-do** loop of Algorithm \mathcal{R} terminates with $InCls = \{13, 14\}$, $Defs = \{D1\}$, and $TransfCls = \{10\}$.

Now, the second iteration starts off by executing the *Diff-Define-Fold* procedure. The procedure handles the two clauses 13 and 14 in $InCls$.

For clause 13, the *Diff-Define-Fold* procedure applies case (Project). Indeed, the body of clause 13 has the following single sharing block:

B_{13} : $\text{reverse}(Zs, Rs), \text{len}(Zs, N1), \text{len}(Rs, N2)$

and there is no clause $\text{newp}(V) \leftarrow d, B'$ in $Defs \cup \text{NewDefs}$ such that B_{13} is an instance of B' . Thus, the procedure adds to NewDef the following new definition (see also Section 2):

Procedure *Replace*(*UnfCls*, *Ds*, *RCls*)
Input: Two sets *UnfCls* and *Ds* of definite clauses;
Output: A set *RCls* of definite clauses.

RCls := *UnfCls*;
repeat
 if there is a clause $C \in RCls$ such that rule R4 is applicable to C
 then $RCls := RCls \setminus \{C\}$;
 if there is a clause $C \in RCls$ such that the Functionality Rule R5 is applicable
 to C with respect to $RCls \cup Ds$, thus deriving a new clause D
 then $RCls := (RCls \setminus \{C\}) \cup \{D\}$;
 if there is a clause $C \in RCls$ such that the Totality Rule R6 is applicable to C
 with respect to $RCls \cup Ds$, thus deriving a new clause D
 then $RCls := (RCls \setminus \{C\}) \cup \{D\}$;
until no rule in {R4, R5, R6} is applicable to a clause in *RCls*

Fig. 4. The *Replace* procedure.

D2. $\text{new2}(N1, N2) :- \text{reverse}(Zs, Rs), \text{len}(Zs, N1), \text{len}(Rs, N2).$

and, by folding clause 13, we get:

15. $\text{new1}(N0, N1, N2) :- N0=0, \text{new2}(N1, N2).$

which has basic types and hence it is added to *FldCls*. This clause is then added to the output set *TransfCls* (see Figure 1).

For clause 14, the *Diff-Define-Fold* procedure applies case (Diff-Introduce). Indeed, the body of clause 14 has the following single sharing block:

$B_{14}: \text{append}(Xs, Ys, Zs), \text{reverse}(Zs, Rs), \text{len}(Xs, N0), \text{len}(Ys, N1),$
 $\text{snoc}(Rs, X, R1s), \text{len}(R1s, N21)$

and we have that $B_1 \lesssim B_{14}$, where B_1 is the body of clause D1, which is the definition introduced as explained in Example 4 above. The procedure constructs the conjunctions defined at Points (i)–(iii) of (Diff-Introduce) as follows:

$M = (\text{append}(Xs, Ys, Zs), \text{reverse}(Zs, Rs), \text{len}(Xs, N0), \text{len}(Ys, N1)),$
 $F(X; Y) = (\text{snoc}(Rs, X, R1s), \text{len}(R1s, N21)),$ where $X = (Rs, X)$, $Y = (R1s, N21)$,
 $R(V; W) = \text{len}(Rs, N2),$ where $V = (Rs)$, $W = (N2).$

In this example, ϑ is the identity substitution. Moreover, the condition on the level mapping ℓ required in the *Diff-Define-Fold* Procedure of Figure 2 is fulfilled because $\ell(\text{new1}) > \ell(\text{snoc})$ and $\ell(\text{new1}) > \ell(\text{len})$. Thus, the definition \hat{D} to be introduced is:

D3. $\text{diff}(X, N2, N21) :- \text{snoc}(Rs, X, R1s), \text{len}(R1s, N21), \text{len}(Rs, N2).$

Indeed, we have that: (i) the projection $\pi(c, X)$ is $\pi(N01=N0+1, (Rs, X))$, that is, the empty conjunction **true**, (ii) $F(X; Y)$, $R(V; W)$ is the body of clause D3, and (iii) the head variables $N2$, X , and $N21$ are the integer variables in that body.

Note that: (i) clause D3 is the one we have presented in Section 2, and (ii) the relationship between the sharing blocks B_1 and B_{14} , which occur in the body of

clauses D1 and 14, respectively, formalizes the notion of mismatch between the bodies of clauses D1 and D1* described in Section 2, because clause 14 is the same as clause D1*.

Then, by applying rule R7 to clause 14, the conjunction ‘ $\text{snoc}(\text{Rs}, \text{X}, \text{R1s}), \text{len}(\text{R1s}, \text{N21})$ ’ can be replaced by the new conjunction ‘ $\text{len}(\text{Rs}, \text{N2}), \text{diff}(\text{X}, \text{N2}, \text{N21})$ ’, and we get the clause:

```
16. new1(N01, N1, N21) :- N01=N0+1, append(Xs, Ys, Zs), reverse(Zs, Rs),
                        len(Xs, N0), len(Ys, N1), len(Rs, N2),
                        diff(X, N2, N21).
```

Finally, by folding clause 16 using clause D1, we get the following clause:

```
17. new1(N01, N1, N21) :- N01=N0+1, new1(N0, N1, N2), diff(X, N2, N21).
```

which has no list arguments and hence it is added to FldCls . This clause is then added to the output set TransfCls .

Algorithm \mathcal{R} proceeds by applying the *Unfold* and *Replace* procedures to clauses D2 and D3. Then, a final execution of the *Diff-Define-Fold* procedure allows us to fold all clauses with ADT terms and derive clauses with basic types, without introducing any new definition. Thus, \mathcal{R} terminates and its output TransfCls is equal (modulo variable renaming) to the set TransfRevCls of clauses listed in Section 2. \square

5.4 Termination of Algorithm \mathcal{R}

As discussed above, each execution of the *Diff-Define-Fold*, *Unfold*, and *Replace* procedures terminates. However, Algorithm \mathcal{R} might not terminate because new predicates may be introduced by *Diff-Define-Fold* at each iteration of the **while-do** of \mathcal{R} , and the loop-exit condition $\text{InCls} \neq \emptyset$ might be never satisfied.

Thus, Algorithm \mathcal{R} terminates if and only if, during its execution, the *Diff-Define-Fold* procedure introduces a finite set of new predicate definitions. A way of achieving this finiteness property is to combine the use of a generalization operator for constraints (see Section 5.2) with a suitable generalization strategy for the conjunctions of atoms that can appear in the body of the definitions (see, for instance, the *most specific generalization* used by *conjunctive partial deduction* [20]). It should be noticed, however, that an effect of a badly designed generalization strategy could be an ADT removal algorithm that often terminates and returns a set of unsatisfiable CHCs whereas the initial clauses were satisfiable (in other terms, the transformation would often generate *spurious counterexamples*).

The study of suitable generalization strategies and also the study of classes of CHCs for which a suitable modification of Algorithm \mathcal{R} terminates are beyond the scope of the present paper. Instead, in Section 8, we evaluate the effectiveness of Algorithm \mathcal{R} from an experimental viewpoint.

5.5 Soundness of Algorithm \mathcal{R}

The soundness of \mathcal{R} follows from the soundness of the transformation rules, and hence we have the following result.

Theorem 5 (Soundness of Algorithm \mathcal{R}). *Suppose that Algorithm \mathcal{R} terminates for an input set Cls of clauses, and let $TransfCls$ be the output set of clauses. Then, every clause in $TransfCls$ has basic types, and if $TransfCls$ is satisfiable, then Cls is satisfiable.*

Proof. Each procedure used in Algorithm \mathcal{R} consists of a sequence of applications of rules R1–R7. Moreover, the Unfold procedure ensures that each clause $H \leftarrow c, B$ introduced by rule R1 is unfolded with respect to an atom A in B such that $\ell(H) = \ell(A)$. Thus, Condition (U) of the hypothesis of Theorem 1 holds for any transformation sequence generated by Algorithm \mathcal{R} , and hence the thesis follows from Theorem 1. \square

6 Preserving Completeness

In the previous Sections 4 and 5, we have shown the soundness of the transformation rules, and hence the soundness of Algorithm \mathcal{R} .

However, the use of rules R1–R7, with the restrictions mentioned in Theorem 1, does *not* preserve completeness, in the sense that, we may construct a transformation sequence $P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_n$, where P_0 is a satisfiable set of clauses and P_n is a set of unsatisfiable clauses. Thus, the hypotheses of Theorem 1 do not guarantee that Algorithm \mathcal{R} preserves completeness. In this section we will introduce some sufficient conditions that guarantee the preservation of completeness.

6.1 Completeness of the transformation rules

Completeness may be affected by the use of rule R3 or rule R7, as shown by the following two examples. In these examples the variables range over the integers \mathbb{Z} or the lists \mathbb{L} of integers, according to their type.

Example 7. Let us consider the following set of clauses:

P_0 :
 1. `false :- Y>0, a([],Y).`
 2. `a([],Y) :- Y=0.`
 3. `a([H|T],Y) :- Y=1.`

We introduce the following clause defining a new predicate by rule R1:

4. `newp(Z) :- a(X,Z).`

and we get $P_1 = \{1, 2, 3, 4\}$. Now, we unfold clause 4 and we derive the clauses:

5. `newp(Y) :- Y=0.`
 6. `newp(Y) :- Y=1.`

We get $P_2 = \{1, 2, 3, 5, 6\}$. Finally, we fold clause 1 using clause 4, which belongs to $Defs_2$, and we derive the set of clauses:

- P_3 :
- 1f. $\text{false} :- Y > 0, \text{newp}(Y)$.
 2. $\text{a}([], Y) :- Y = 0$.
 3. $\text{a}([H|T], Y) :- Y = 1$.
 5. $\text{newp}(Y) :- Y = 0$.
 6. $\text{newp}(Y) :- Y = 1$.

Now, we have that P_0 is satisfiable (because $\text{a}([], Y)$ holds for $Y=0$ only), while P_3 is unsatisfiable (because $\text{newp}(Y)$ holds for $Y=1$).

Let us explain why, in this example, folding affects completeness. By applying the Folding Rule R3 to clause 1, we have replaced atom $\text{a}([], Y)$ by atom $\text{newp}(Y)$, which, by clause 4, is equivalent to $\exists X. \text{a}(X, Y)$ (because X does not occur in the head of clause 4). Thus, folding is based on a substitution for the existentially quantified variable X which is not the identity. Now in $M(P_1)$, which is $\{\text{a}([], 0), \text{a}([h|t], 1), \text{newp}(0), \text{newp}(1) \mid h \in \mathbb{Z}, t \in \mathbb{L}\}$, atoms $\text{a}([], Y)$ and $\text{newp}(Y)$ are *not equivalent*. Indeed, $M(P_1) \models \text{a}([], Y) \rightarrow \text{newp}(Y)$, while $M(P_1) \not\models \text{newp}(1) \rightarrow \text{a}([], 1)$. \square

Example 8. Let us consider the following set of clauses:

- P_0 :
1. $\text{false} :- Y > 0, \text{a}(X), \text{f}(X, Y)$.
 2. $\text{a}([])$.
 3. $\text{f}([], Y) :- Y = 0$.
 4. $\text{f}([H|T], Y) :- Y = 1$.
 5. $\text{r}(X, W) :- W = 1$.

We introduce the following clause defining the new predicate diff by rule R1:

6. $\text{diff}(W, Y) :- \text{f}(X, Y), \text{r}(X, W)$.

where: (i) $\text{f}(X, Y)$ is a total, functional atom from X to Y , and (ii) $\text{r}(X, W)$ is a total, functional atom from X to W . Thus, we get $P_1 = \{1, 2, 3, 4, 5, 6\}$ and $\text{Defs}_1 = \{6\}$. By applying the Differential Replacement Rule R7, from P_1 we derive the following set of clauses:

- P_2 :
- 1r. $\text{false} :- Y > 0, \text{a}(X), \text{r}(X, W), \text{diff}(W, Y)$.
 2. $\text{a}([])$.
 3. $\text{f}([], Y) :- Y = 0$.
 4. $\text{f}([H|T], Y) :- Y = 1$.
 5. $\text{r}(X, W) :- W = 1$.
 6. $\text{diff}(W, Y) :- \text{f}(X, Y), \text{r}(X, W)$.

Now, we have that P_0 is satisfiable (because $\text{a}(X)$ holds for $X=[]$ only, and $\text{f}([], Y)$ holds for $Y=0$ only), while P_2 is unsatisfiable (because the body of clause 1r holds for $X=[]$ and $W=Y=1$).

Let us now explain why, in this example, the application of rule R7 affects completeness. By applying rule R7, we have replaced atom $\text{f}(X, Y)$ by the conjunction ' $\text{r}(X, W), \text{diff}(W, Y)$ ', which by clause 6 and the totality of $\text{r}(X, W)$ from X to W , is implied by $\text{f}(X, Y)$. However, in $M(P_1)$, which is $\{\text{a}([], 0), \text{f}([], 0), \text{f}([h|t], 1), \text{r}(u, 1), \text{diff}(1, 0), \text{diff}(1, 1) \mid h \in \mathbb{Z}, t, u \in \mathbb{L}\}$, atom $\text{f}(X, Y)$ and the conjunction ' $\text{r}(X, W), \text{diff}(W, Y)$ ' are *not equivalent*. Indeed, we have that $M(P_1) \not\models (\text{r}([], 1) \wedge \text{diff}(1, 1)) \rightarrow \text{f}([], 1)$.

In particular, note that when applying rule R7, we have replaced $\text{f}(X, Y)$, which is functional from X to Y , by ' $\text{r}(X, W), \text{diff}(W, Y)$ ', which is not functional

from \mathbf{x} to (\mathbf{w}, \mathbf{y}) (indeed, for all $\mathbf{u} \in \mathbb{L}$, we have that both ‘ $\mathbf{r}(\mathbf{u}, \mathbf{1})$, $\mathbf{diff}(\mathbf{1}, \mathbf{0})$ ’ and ‘ $\mathbf{r}(\mathbf{u}, \mathbf{1})$, $\mathbf{diff}(\mathbf{1}, \mathbf{1})$ ’ do hold). This is due to the fact that $\mathbf{diff}(\mathbf{w}, \mathbf{y})$ is *not functional* from \mathbf{w} to \mathbf{y} . \square

Now, we will give some sufficient conditions that guarantee that the transformation rules presented in Section 4 are complete, in the sense that if a set P_0 of CHCs is transformed into a new set P_n by n applications of the rules and P_0 is satisfiable, then also P_n is satisfiable. Thus, when those conditions hold, the converse of the Soundness Theorem 1 holds.

We consider the following Conditions (E) and (F) on the application of the Folding Rule R3 [23, 51] and the Differential Replacement Rule R7, respectively.

Definition 9 (Condition E). Let us assume that: (i) we apply the Folding Rule R3 for folding clause $C: H \leftarrow c, G_L, Q, G_R$ in P_i using the definition $D: K \leftarrow d, B$, and (ii) ϑ is a substitution such that $Q = B\vartheta$ and $\mathbb{D} \models \forall(c \rightarrow d\vartheta)$. We say that this application of rule R3 *fulfills Condition (E)* if the following holds:

- (E) for every variable $X \in \text{vars}(\{d, B\}) \setminus \text{vars}(K)$,
- E1. $X\vartheta$ is a variable not occurring in $\{H, c, G_L, G_R\}$ and
 - E2. $X\vartheta$ does not occur in the term $Y\vartheta$, for any variable Y occurring in (d, B) and different from X .

In Condition (F) below, we consider the particular case, which is of our interest in this paper, when rule R7 is applied within the *Diff-Define-Fold* procedure.

Definition 10 (Condition F). Let us assume that we apply the Differential Replacement Rule R7 to clause $C: H \leftarrow c, G_L, F(X; Y), G_R$ in P_i using the definition $\hat{D}: \mathbf{diff}(T_b, W_b, Y_b) \leftarrow d, F(X; Y), R(V; W)$ in Defs_i , where $T_b = \text{bvars}(X \cup V)$, $W_b = \text{bvars}(W)$, and $Y_b = \text{bvars}(Y)$. We say that this application of rule R7 *fulfills Condition (F)* if the following holds:

- (F) F1. atom $\mathbf{diff}(T_b, W_b, Y_b)$ is functional from (T_b, W_b) to Y_b with respect to $\text{Definite}(P_0) \cup \text{Defs}_i$,
- F2. $Y \cap (V \cup \text{vars}(d)) = \emptyset$, and
 - F3. $\text{adt-vars}(Y) \cap \text{adt-vars}(\{H, c, G_L, G_R\}) = \emptyset$.

The following theorem guarantees that, if Conditions (E) and (F) hold, then the transformation rules R1–R7 are complete.

Theorem 6 (Completeness of the Transformation Rules). *Let $P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_n$ be a transformation sequence using rules R1–R7. Suppose that, for every application of R3, Condition (E) holds, and for every application of R7, Condition (F) holds. If P_0 is satisfiable, then P_n is satisfiable.*

Note that the applications of R3 and R7 in Examples 7 and 8 violate Conditions (E) and (F), respectively, and these facts explain why they affect completeness. Indeed, in Example 7, atom $\mathbf{a}(\square, \mathbf{y})$ in the body of clause 1 is an instance of the body of clause 4 via the substitution $\vartheta = \{\mathbf{x}/\square, \mathbf{z}/\mathbf{y}\}$, and ϑ does not satisfy Condition (E1). In Example 8, as mentioned above, $\mathbf{diff}(\mathbf{w}, \mathbf{y})$ is not functional from \mathbf{w} to \mathbf{y} , and hence Condition (F1) is not fulfilled.

For the proof of Theorem 6 we need some preliminary results. First we prove the following theorem, which is the converse of Theorem 4 and is a consequence of Theorem 3 reported in Section 4.2.

Theorem 7. *Let $P_0 \Rightarrow \dots \Rightarrow P_n$ be a transformation sequence constructed using rules R1 (Definition), R2 (Unfolding), R3 (Folding), and R8 (Goal Replacement). Suppose that, for all applications of R3, Condition (E) holds and all goal replacements are body strengthenings (that is, they are applications of rule R8 for which Condition (S) of Section 4.2 holds). If P_0 is satisfiable, then P_n is satisfiable.*

Proof. As shown in the proof of Theorem 4, P_0 is satisfiable iff $P_0 \cup \text{Defs}_n$ is satisfiable. Also in this proof, we consider the transformation sequence $P'_0 \Rightarrow \dots \Rightarrow P'_n$ obtained from the sequence $P_0 \Rightarrow \dots \Rightarrow P_n$ by replacing each occurrence of *false* in the head of a clause by a new predicate symbol f . P'_0, \dots, P'_n are sets of definite clauses, and thus for $i = 0, \dots, n$, $\text{Definite}(P'_i) = P'_i$. The sequence $P'_0 \Rightarrow \dots \Rightarrow P'_n$ satisfies the hypotheses of Theorem 3, and hence $M(P'_0 \cup \text{Defs}_n) \supseteq M(P'_n)$. Thus, we have that:

P_0 is satisfiable
implies $P_0 \cup \text{Defs}_n$ is satisfiable
implies $P'_0 \cup \text{Defs}_n \cup \{\neg f\}$ is satisfiable
implies $f \notin M(P'_0 \cup \text{Defs}_n)$
implies, by Theorem 3, $f \notin M(P'_n)$
implies $P'_n \cup \{\neg f\}$ is satisfiable
implies P_n is satisfiable. □

Now, in order to prove Theorem 6 of Section 4, we show that rules R4–R7 are all body strengthenings.

Rule R4 (Clause Deletion) is a body strengthening, as

$$M(\text{Definite}(P_0) \cup \text{Defs}_i) \models \forall (\text{false} \rightarrow c \wedge G)$$

trivially holds.

Now let us consider rule R5 (Functionality). Let $F(X, Y)$ be a conjunction of atoms that defines a functional relation from X to Y . When rule R5 is applied whereby the conjunction $F(X, Y)$, $F(X, Z)$ is replaced by the conjunction $Y = Z$, $F(X, Y)$, it is the case that

$$M(\text{Definite}(P_0) \cup \text{Defs}_i) \models \forall (Y = Z \wedge F(X, Y) \rightarrow F(X, Y) \wedge F(X, Z))$$

Hence, Condition (S) holds and rule R5 is a body strengthening.

An application of rule R6 (Totality) replaces a conjunction $F(X, Y)$ by *true* (that is, the empty conjunction). When rule R6 is applied, it is the case that, by Property (*Total*) of Section 3,

$$M(\text{Definite}(P_0) \cup \text{Defs}_i) \models \forall (\text{true} \rightarrow \exists Y. F(X, Y))$$

Hence, rule R6 is a body strengthening.

Now we prove that, when Condition (F) of Definition 10 holds, rule R7 is a body strengthening.

Lemma 3. *Let us consider the following clauses C , \widehat{D} , and E used when applying rule R7:*

$$\begin{aligned} C: & H \leftarrow c, G_L, F(X; Y), G_R \\ \widehat{D}: & \text{diff}(T_b, W_b, Y_b) \leftarrow d, F(X; Y), R(V; W) \\ E: & H \leftarrow c, G_L, R(V; W), \text{diff}(T_b, W_b, Y_b), G_R \end{aligned}$$

where $Y = (Y_a, Y_b)$, $Y_a = \text{adt-vars}(Y)$, and $Y_b = \text{bvars}(Y)$. Let us assume that Conditions (F1) and (F2) of Definition 10 hold. Then,

$$M(\text{Definite}(P_0) \cup \text{Defs}_i) \models \forall (c \wedge R(V; W) \wedge \text{diff}(T_b, W_b, Y_b) \rightarrow \exists Y_a. c \wedge F(X; Y)).$$

Proof. Let \mathcal{M} denote $M(\text{Definite}(P_0) \cup \text{Defs}_i)$. Let $Y' = (Y_a, Y'_b)$, where Y'_b is obtained by renaming the variables in Y_b with new variables of basic type. By the totality of $F(X; Y')$, we have:

$$\mathcal{M} \models \forall (c \wedge R(V; W) \wedge \text{diff}(T_b, W_b, Y_b) \rightarrow \exists Y'. F(X; Y'))$$

By the definition of rule R7, $\mathbb{D} \models \forall (c \rightarrow d)$ holds, and we get:

$$\mathcal{M} \models \forall (c \wedge R(V; W) \wedge \text{diff}(T_b, W_b, Y_b) \rightarrow \exists Y'. d \wedge F(X; Y') \wedge R(V; W))$$

Now, we have that $Y \cap (X \cup V \cup W \cup \text{vars}(d)) = \emptyset$. Indeed, (i) by the definition of rule R7, $W \cap \text{vars}(C) = \emptyset$, (ii) by the notation ' $F(X; Y)$ ', we have that $\text{vars}(X) \cap \text{vars}(Y) = \emptyset$, and (iii) by Condition (F2), $Y \cap (V \cup \text{vars}(d)) = \emptyset$. Then, $d \wedge F(X; Y') \wedge R(V; W)$ is a variant of the body of clause \widehat{D} , and since $Y' = (Y_a, Y'_b)$, we get:

$$\begin{aligned} \mathcal{M} \models \forall (c \wedge R(V; W) \wedge \text{diff}(T_b, W_b, Y_b) \\ \rightarrow \exists Y_a, Y'_b. F(X; (Y_a, Y'_b)) \wedge \text{diff}(T_b, W_b, Y'_b)) \end{aligned}$$

By Condition (F1), $\text{diff}(T_b, W_b, Y_b)$ is functional from (T_b, W_b) to Y_b , and we have:

$$\mathcal{M} \models \forall (c \wedge R(V; W) \wedge \text{diff}(T_b, W_b, Y_b) \rightarrow \exists Y_a, Y'_b. F(X; (Y_a, Y'_b)) \wedge Y_b = Y'_b)$$

Thus,

$$\mathcal{M} \models \forall (c \wedge R(V; W) \wedge \text{diff}(T_b, W_b, Y_b) \rightarrow \exists Y_a. F(X; (Y_a, Y_b)))$$

and, observing that $Y_a \cap \text{vars}(c) = \emptyset$, we get the thesis. \square

Now, in order to show that an application of rule R7 according to the hypotheses of Lemma 3 (see also Definition 10) is an instance of a body strengthening, where in clauses C and D of rule R8 we consider $c = \text{true}$ and $c_1 = c_2 = c$, we have to show:

$$(S_c) \quad M(\text{Definite}(P_0) \cup \text{Defs}_i) \models \forall (c \wedge G_2 \rightarrow \exists T_1. c \wedge G_1)$$

where:

$$\begin{aligned} T_1 &= \text{vars}(c \wedge F(X; (Y_a, Y_b))) \setminus \text{vars}(\{H, \text{true}, G_L, G_R\}) \\ G_1 &= F(X; (Y_a, Y_b)), \text{ and} \\ G_2 &= R(V; W), \text{diff}(T_b, W_b, Y_b). \end{aligned}$$

Now, by Lemma 3, we have:

$$(L) \quad M(\text{Definite}(P_0) \cup \text{Defs}_i) \models \forall (c \wedge G_2 \rightarrow \exists Y_a. c \wedge F(X; (Y_a, Y_b)))$$

Since: (i) by Condition (F3) and the fact that the variables of c are all of basic type, we have that: $Y_a \subseteq T_1 = \text{vars}(c \cup X \cup Y_a \cup Y_b) \setminus \text{vars}(\{H, \text{true}, G_L, G_R\})$,

and (ii) the variables in $T_1 \setminus Y_a$ are universal quantified in (L), we have that (L) implies (S_c). This completes the proof that if Condition (F) of Definition 10 holds, then rule R7 is a body strengthening.

Thus, by taking into account also the facts we have proved above about rules R4, R5, and R6, we have the following lemma.

Lemma 4. *All the applications of rules R4, R5, R6, and the applications of rule R7 where Condition (F) holds (see Definition 10), are body strengthenings.*

Now we can present the proof of Theorem 6.

Proof of Theorem 6. Let $P_0 \Rightarrow \dots \Rightarrow P_n$ be a transformation sequence using rules R1–R7. Suppose that, for every application of R3, Condition (E) holds, and for every application of R7, Condition (F) holds. Thus, $P_0 \Rightarrow \dots \Rightarrow P_n$ can also be constructed by applications of rules R1–R3 and applications of rule R8 which, by Lemma 4, are all body strengthenings. Then, the thesis follows from Theorem 7. \square

6.2 Completeness of Algorithm \mathcal{R}

We have the following straightforward consequence of Theorem 6.

Theorem 8 (Completeness of Algorithm \mathcal{R}). *Suppose that Algorithm \mathcal{R} terminates for the input set Cls of clauses, and let TransfCls be the output set of clauses. Suppose also that all applications of rules R3 and R7 during the execution of Algorithm \mathcal{R} fulfill Conditions (E) and (F), respectively. If Cls is satisfiable, then TransfCls is satisfiable.*

In practice, having constructed the transformation sequence $P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_n$, where rule R7 has been applied to the set P_i , with $0 < i < n$, it is often more convenient to check the validity of Condition (F) with respect to $Definite(P_n)$, instead of $Definite(P_0) \cup Defs_i$, as required by the hypotheses of Theorem 6. Indeed, in the set P_n , predicate *diff* is defined by a set of clauses whose variables have all integer or boolean type, and hence in checking Condition (F) we need not reason about predicates defined over ADTs. The following Proposition 1 guarantees that Theorem 6 holds even if we check Condition (F) with respect to $Definite(P_n)$, instead of $Definite(P_0) \cup Defs_i$.

Proposition 1 (Preservation of Functionality). *Let $P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_n$ be a transformation sequence using rules R1–R7. Suppose that Condition (U) of Theorem 1 holds. For $i = 0, \dots, n$, if an atom $A(X, Y)$ is functional from X to Y with respect to $Definite(P_n)$ and the predicate symbol of $A(X, Y)$ occurs in $Definite(P_0) \cup Defs_i$, then $A(X, Y)$ is functional from X to Y with respect to $Definite(P_0) \cup Defs_i$.*

Proof. Let us suppose that $A(X, Y)$ is functional from X to Y with respect to $Definite(P_n)$, that is,

$$M(\text{Definite}(P_n)) \models \forall (A(X, Y) \wedge A(X, Z) \rightarrow Y = Z).$$

Then, for all (tuples of) ground terms u, v , and w , with $v \neq w$,

$$\{A(u, v), A(u, w)\} \not\subseteq M(\text{Definite}(P_n))$$

Since $P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_n$ is a transformation sequence using rules R1–R7 such that Condition (U) of Theorem 1 holds, then by Theorem 2,

$$\{A(u, v), A(u, w)\} \not\subseteq M(\text{Definite}(P_0) \cup \text{Defs}_n)$$

Hence, for $i=0, \dots, n$,

$$\{A(u, v), A(u, w)\} \not\subseteq M(\text{Definite}(P_0) \cup \text{Defs}_i)$$

Thus,

$$M(\text{Definite}(P_0) \cup \text{Defs}_i) \models \forall (A(X, Y) \wedge A(X, Z) \rightarrow Y = Z). \quad \square$$

7 A Method for Checking the Satisfiability of CHCs through ADT Removal

In this section we put together the results presented in Sections 4, 5, and 6 and we define a method for checking whether or not a set P_0 of CHCs is satisfiable. We proceed as follows: (i) first, we construct a transformation sequence $P_0 \Rightarrow P_1 \Rightarrow \dots \Rightarrow P_n$ using Algorithm \mathcal{R} , and (ii) then we apply a CHC solver to P_n . If the solver is able to prove the satisfiability of P_n , then, by Theorem 5, P_0 is satisfiable. If the solver proves the unsatisfiability of P_n and Conditions (E) and (F) are both fulfilled during the execution of \mathcal{R} , then, by Theorem 8, P_0 is unsatisfiable.

Now, (i) Condition (E) can be checked by simply inspecting the substitution computed when applying the Folding Rule R3 during the *Diff-Define-Fold* procedure. (ii) Condition (F1), by Proposition 1, can be checked by proving, for every difference predicate *diff* that has been used in applying the Differential Replacement Rule R7, the satisfiability of the following set of clauses

$$D_n \cup \{\text{false} \leftarrow Y_1 \neq Y_2, \text{diff}(T, W, Y_1), \text{diff}(T, W, Y_2)\},$$

where D_n is the set of clauses defining *diff* in $\text{Definite}(P_n)$. (iii) Finally, Conditions (F2) and (F3) can be checked by inspecting, for every application of rule R7, the clauses C and \hat{D} involved in that application (see Definition 10).

In previous sections we have seen in action our method for proving the satisfiability of a set of CHCs. In the following example we show an application of our method to prove unsatisfiability of a set of CHCs.

Example 9. Let us consider again our introductory example of Section 2 where we started from the initial set *RevCls* made out of clauses 1–9. Let us suppose that we want to check the satisfiability of the set *RevCls** of clauses that includes clauses 2–9 and the following clause 1*, instead of clause 1:

```
1*. false :- N2=\=N0-N1, append(Xs,Ys,Zs), reverse(Zs,Rs),
           len(Xs,N0), len(Ys,N1), len(Rs,N2).
```

Clause 1^* differs from clause 1 because of the constraint ' $N2=\backslash=N0-N1$ ', instead of ' $N2=\backslash=N0+N1$ '. The set $RevCls^*$ is unsatisfiable because the body of clause 1^* holds, in particular, for $Xs=[]$ and $Ys=[Y]$, where Y is any integer.

Algorithm \mathcal{R} works for the input set $RevCls^* = \{1^*, 2, \dots, 9\}$ exactly as described in Section 5 for the set $RevCls = \{1, 2, \dots, 9\}$, except that clause 1^* , instead of clause 1, is folded by using the definition:

D1. $new1(N0, N1, N2) :- \text{append}(Xs, Ys, Zs), \text{reverse}(Zs, Rs), \text{len}(Xs, N0),$
 $\text{len}(Ys, N1), \text{len}(Rs, N2).$

thereby deriving the following clause, instead of clause 10:

$10^*. \text{false} :- N2=\backslash=N0-N1, new1(N0, N1, N2).$

Thus, the output of Algorithm \mathcal{R} is $TransfRevCls^* = \{10^*, 15, 17, 18, 19, 20, 21\}$ (for clauses 15, 17, 18, 19, 20, 21 see Section 2). The CHC solver Eldarica proves that $TransfRevCls^*$ is an unsatisfiable set of clauses.

Now, in order to conclude that also the input set $RevCls^*$ is unsatisfiable, we apply Theorem 8 and Proposition 1. We look at the transformation sequence constructed by Algorithm \mathcal{R} and we check that both Conditions (E) and (F) are fulfilled.

Condition (E) is fulfilled because each time we apply the folding rule, the substitution ϑ is the identity (see, in particular, the folding step for deriving clause 10^* above, and also the folding steps in Example 6).

Now, let us check Condition (F). We have that clauses C and \widehat{D} occurring in Definition 10 are clauses 14 and D3, respectively. For the reader's convenience, we list them here:

14. $new1(N01, N1, N21) :- N01=N0+1, \text{append}(Xs, Ys, Zs), \text{reverse}(Zs, Rs),$
 $\text{len}(Xs, N0), \text{len}(Ys, N1), \text{snoc}(Rs, X, R1s),$
 $\text{len}(R1s, N21).$

D3. $\text{diff}(X, N2, N21) :- \text{snoc}(Rs, X, R1s), \text{len}(R1s, N21), \text{len}(Rs, N2).$

With reference to Definition 10 (see Example 6), we have:

$F(X; Y) = (\text{snoc}(Rs, X, R1s), \text{len}(R1s, N21)), \quad X = (Rs, X), \quad Y = (R1s, N21),$
 $R(V; W) = \text{len}(Rs, N2), \quad V = (Rs), \quad W = (N2).$

Condition (F1) requires that atom $\text{diff}(X, N2, N21)$ be functional from $(X, N2)$ to $N21$ with respect to $Definite(RevCls^*)$. In order to check this functionality, by Proposition 1, it suffices to check the satisfiability of the set consisting of following clause:

22. $\text{false} :- N21=\backslash=N22, \text{diff}(X, N2, N21), \text{diff}(X, N2, N22).$

together with clauses 20 and 21, which define diff in $Definite(TransfRevCls^*)$. We recall them here:

20. $\text{diff}(X, N0, N1) :- N0=0, N1=1.$

21. $\text{diff}(X, N0, N1) :- N0=N+1, N1=M+1, \text{diff}(X, N, M).$

The CHC solver Eldarica proves the satisfiability of the set $\{20, 21, 22\}$ of clauses by computing the following model:

$\text{diff}(X, N2, N21) :- N21=N2+1, N2>=0.$

Thus, Condition (F1) is fulfilled. Condition (F2) requires that $\{R1s, N21\} \cap \{Rs\} = \emptyset$, and Condition (F3) requires that $\{R1s\} \cap \{Xs, Ys, Zs, Rs\} = \emptyset$. They

are both fulfilled. Therefore, by Theorem 8, we conclude, as desired, that the input set $RevCls^*$ of clauses is unsatisfiable. \square

8 Experimental Evaluation

In this section we present the experimental evaluation we have performed for assessing the effectiveness of our transformation-based CHC satisfiability checking method.

We have implemented our method in a tool called ADTREM and we have compared the results obtained by running our tool with those obtained by running: (i) the CVC4 SMT solver [46] extended with inductive reasoning and lemma generation, (ii) the AdtInd solver [54], which makes use of a syntax-guided synthesis strategy [1] for lemma generation, and (iii) the Eldarica CHC solver [29], which combines predicate abstraction [26] with counterexample-guided abstraction refinement [9]. ADTREM is available at <https://fmlab.unich.it/adtrem/>.

8.1 The workflow of the ADTREM tool

Our ADTREM tool implements the satisfiability checking method presented in Section 7 as follows. First, ADTREM makes use of the VeriMAP system [11] to perform the steps specified by Algorithm \mathcal{R} . It takes as input a set P_0 of CHCs and, if it terminates, it produces as output a set P_n of CHCs that have basic types. Then, in order to show the satisfiability of P_0 , ADTREM invokes the Eldarica CHC solver to show the satisfiability of P_n .

If Eldarica proves that P_n is satisfiable, then, by the soundness of the transformation Algorithm \mathcal{R} (see Theorem 5), P_0 is satisfiable and ADTREM returns the answer ‘*sat*’. In particular, the implementation of the *Unfold* procedure enforces Condition (U) of Theorem 1, which indeed ensures the soundness of Algorithm \mathcal{R} .

If Eldarica proves that P_n is unsatisfiable by constructing a counterexample CEX, ADTREM proceeds by checking whether or not Conditions (E) and (F), which guarantee the completeness of Algorithm \mathcal{R} , hold (see Theorem 8). In particular, during the execution of Algorithm \mathcal{R} , ADTREM checks Condition (E) when applying the Folding Rule R3, and Conditions (F2) and (F3) when applying the Differential Replacement Rule R7. ADTREM marks all clauses in P_n derived by a sequence of transformation steps where one of the Conditions (E), (F2), and (F3) is not satisfied.

Then, ADTREM looks at the counterexample CEX constructed by Eldarica to verify whether or not any instance of the marked clauses is used for the construction of CEX. If this is the case, it is not possible to establish the unsatisfiability of P_0 from the unsatisfiability of P_n (as completeness of Algorithm \mathcal{R} may not hold) and hence ADTREM returns the answer ‘*unknown*’ as the result of the satisfiability check for P_0 .

Otherwise, if no instance of a marked clause is used in CEX, then ADTREM proceeds by checking Condition (F1) of Definition 10. Recall that, by Proposition 1, Condition (F1) can be checked by inspecting the clauses defining the

differential predicates in $Definite(P_n)$. To perform this check, for each differential predicate $diff_k$ introduced by Algorithm \mathcal{R} and occurring in CEX, ADTREM produces a clause, call it $fun\text{-}diff_k$, of the form:

$$false \leftarrow O_1 \neq O_2, diff_k(I, O_1), diff_k(I, O_2)$$

where I and O_i , for $i = 1, 2$, are the tuples of input and output variables, respectively, of $diff_k(I, O_i)$. Then, ADTREM runs Eldarica to check the satisfiability of $\bigcup_k \{fun\text{-}diff_k\} \cup D_n$, where D_n is the set of clauses defining $diff_k$ in $Definite(P_n)$ (see Example 9). Now, this satisfiability check may succeed (Case 1) or may not succeed (Case 2). In Case 1, Condition (F1) holds and, by the completeness of Algorithm \mathcal{R} (see Theorem 8), we have that P_0 is unsatisfiable and ADTREM returns the answer ‘*unsat*’. In Case 2, Condition (F1) cannot be shown, and ADTREM returns the answer ‘*unknown*’ as the result of the satisfiability of P_0 .

8.2 Benchmark suite

Our benchmark suite consists of 251 verification problems over inductively defined data structures, such as lists, queues, heaps, and trees. Out of these 251 problems, 168 of them refer to properties that hold (*valid* properties) and the remaining 83 refer to properties that do not hold (*invalid* properties).

The 168 problems specifying valid properties have been adapted from the benchmark suite considered by Reynolds and Kuncak [46], and originate from benchmarks used by various theorem provers, such as CLAM [30], HipSpec [8], IsaPlanner [21, 32], and Leon [50]. In particular, we have considered Reynolds and Kuncak’s ‘*dt*’ encoding where natural numbers are represented using the built-in SMT type *Int*. From those problems we have discarded: (i) the ones that do not use ADTs, and (ii) the ones that cannot be directly represented in Horn clause format. In order to make a comparison between our approach and Reynolds-Kuncak’s one on a level playing field, since ADTREM supports neither higher order functions nor user-provided lemmas, (i) we have replaced higher order functions by suitable first order instances, and (ii) we have removed all auxiliary lemmas from the formalization of the problems. We have also used LIA constraints, instead of the basic functions recursively defined over natural numbers, such as the *plus* function and *less-or-equal* relation, so that the solver can deal with them by using the LIA theory.

The 83 problems specifying invalid properties have been obtained from those specifying valid properties by either negating the properties or modifying the definitions of the predicates on which the properties depend.

The benchmark suite is available at <https://fmlab.unich.it/adtrem/>.

8.3 Experiments

We have performed the following experiments.

1. We have run the ‘*cvc4+ig*’ configuration of the CVC4 solver extended with inductive reasoning and also the AdtInd solver on the 251 verification problems in SMT-LIB format.

2. Then, we have translated each verification problem into a set, call it P_0 , of CHCs in the Prolog-like syntax supported by ADTREM by using a modified version of the SMT-LIB parser of the ProB system [39]. We have run Eldarica v2.0.5, which uses no induction-based mechanism for handling ADTs, to check the satisfiability of the SMT-LIB translation of P_0 ⁵.
3. Finally, we have run ADTREM to check the satisfiability of P_0 . If ADTREM returns ‘*sat*’, the property specified by P_0 is reported to be valid. If ADTREM returns ‘*unsat*’, the property is reported to be invalid.

Experiments have been performed on an Intel Xeon CPU E5-2640 2.00GHz with 64GB RAM under CentOS and for each problem we have set a timeout limit of 300 seconds.

8.4 Evaluation of Results

The results of our experiments are summarized in the following four tables.

- In Table 1 (*Solved problems*) we report the number of problems solved by each tool, also classified by the type of property (valid or invalid). Columns 3–6 are labeled by the name of the tool and the last column reports the results of the ‘Virtual Best’ tool, that is, the number of problems solved by at least one of the tools we have considered.
- In Table 2 (*Uniquely solved problems*) we report, for each tool, the number of uniquely solved problems, that is, the number of problems solved by that tool and not solved by any of the other tools. By definition, there are no problems uniquely solved by the ‘Virtual Best’ tool.
- In order to assess the difficulty of the benchmark problems, we have computed, for each problem, the number of tools that are able to solve it. The results are reported in Table 3 (*Benchmark difficulty*).
- In Table 4 (*Termination of Algorithm \mathcal{R} and effectiveness of ADTREM*) we report the number of problems for which the ADT removal algorithm \mathcal{R} terminates and the percentage of those problems solved by ADTREM.

Type of properties	Number of problems	CVC4 with induction	AdtInd	Eldarica	ADTREM	Virtual Best
Valid	168	75	62	12	115	127
Invalid	83	0	1	75	61	83
Total	251	75	63	87	176	210

Table 1. *Solved problems.* Number of problems solved by each tool.

Table 1 shows that, on our benchmark, ADTREM compares favorably to all other tools we have considered.

⁵ We have also performed analogous experiments on the set of valid properties by using Z3-SPACER [35], instead of Eldarica, as reported on an earlier version of this paper [18]. The results of those experiments, which we do not report here, are very similar to those shown in Table 1.

On problems with valid properties, ADTREM performs better than solvers extended with inductive reasoning, such as CVC4 and AdtInd. ADTREM performs better than those two tools also on problems with invalid properties on which CVC4 and AdtInd show poor results. This poor outcome may be due to the fact that those tools were designed with the aim of proving theorems rather than finding counterexamples to non-theorems.

Table 1 also shows that the ADT removal performed by Algorithm \mathcal{R} , implemented by ADTREM, considerably increases the overall effectiveness of the CHC solver Eldarica, without the need for any inductive reasoning support. In particular, Eldarica is able to solve 87 problems out of 251 *before* the application of Algorithm \mathcal{R} (see Column ‘Eldarica’), while ADTREM solves 176 problems by using Eldarica *after* the application of Algorithm \mathcal{R} (see Column ‘ADTREM’).

The gain in effectiveness is very high on problems with valid properties, where Eldarica solves 12 problems out of 168, while ADTREM solves 115 problems by applying Eldarica after the removal of ADTs.

On problems with invalid properties Eldarica is already very effective before the removal of ADTs and is able to solve 75 problems out of 83, whereas the number of problems solved by ADTREM is only 61, which are all the problems with invalid properties for which Algorithm \mathcal{R} terminates (see Table 4). Note, however, that by inspecting the detailed results of our experiments (see <https://fmrlab.unich.it/adtreml/>), we have found 8 problems with invalid properties solved by ADTREM, which are not solved by Eldarica before ADT removal.

Type of properties	Number of problems	CVC4 with induction	AdtInd	Eldarica	ADTREM	Virtual Best
Valid	50	3	2	1	44	-
Invalid	29	0	0	22	7	-
Total	79	3	2	23	51	-

Table 2. *Uniquely solved problems.* Number of uniquely solved problems by each tool.

Table 2 shows further evidence that the overall performance of ADTREM is higher than that of the other tools. Indeed, the number of problems solved by ADTREM only is larger than the number of problems uniquely solved by any other tool. In particular, ADTREM uniquely solves 51 problems (44 with valid properties, 7 with invalid properties) and Eldarica uniquely solves 23 problems (1 with valid property, 22 with invalid properties).

Table 3 illustrates the degree of difficulty of the problems of the benchmark for the tools we have considered. Indeed, out of the 251 problems in the benchmark, 41 of them are solved by no tool, and only 8 problems are solved by all tools.

Table 4 shows that Algorithm \mathcal{R} terminates quite often and, whenever it terminates Eldarica is able to check the satisfiability of the derived set of clauses

Type of properties	Number of problems	Unsolved	Uniquely solved	Solved by two tools	Solved by three tools	Solved by all tools
Valid	168	41	50	25	44	8
Invalid	83	0	29	54	0	0
Total	251	41	79	79	44	8

Table 3. *Benchmark difficulty.* Number of problems grouped by the number of tools that were able to solve them.

Type of properties	Number of problems	Algorithm \mathcal{R} terminates	Percentage solved by ADTREM
Valid	168	117	98%
Invalid	83	61	100%
Total	251	178	99%

Table 4. *Termination of Algorithm \mathcal{R} and effectiveness of ADTREM.* Number of problems for which the ADT removal algorithm terminates and the percentage of those problems solved by ADTREM.

in almost all cases. Indeed, Algorithm \mathcal{R} terminates on 178 problems out of 251, and ADTREM solves 176 problems out of those 178.

Note that the use of the Differential Replacement Rule R7 (which is a novel rule we have used in this paper) has a positive effect on the termination of Algorithm \mathcal{R} . In order to assess this effect, we have implemented a modified version of the ADT removal algorithm \mathcal{R} , called \mathcal{R}° , which *does not* introduce difference predicates. Indeed, in \mathcal{R}° the *Diff-Introduce* case of the *Diff-Define-Fold* Procedure of Figure 2 is never executed. We have applied Algorithm \mathcal{R}° to the 168 problems with valid properties and it terminated only on 94 of them, while \mathcal{R} terminates on 117 (see Table 4). Details are given in <https://fmlab.unich.it/adtrem/>.

The effectiveness of the solvers that use induction we have considered, namely, CVC4 and AdtInd, may depend on the supply of suitable lemmas to be used for proving the main conjecture and also on the representation of the natural numbers. Indeed, further experiments we have performed (see <https://fmlab.unich.it/adtrem/>) show that, on problems with valid properties, CVC4 and AdtInd solve 102 (instead of 75) and 64 (instead of 62) problems, respectively, when auxiliary lemmas are added as extra axioms. If, in addition, we consider the ‘dti’ encoding of the natural numbers⁶, CVC4 and AdtInd solve 139 and 59 problems, respectively. Our results show (see Table 1) that in most cases ADTREM needs neither those extra axioms nor that sophisticated encoding.

Finally, in Table 5 we report some problems with valid properties solved by ADTREM that are not solved by CVC4 with induction, nor by AdtInd, nor by

⁶ In the ‘dti’ encoding, natural numbers are represented using both the built-in type *Int* and the ADT inductive definition with the zero and successor constructors [46].

Eldarica. CVC4 with induction and AdtInd are not able to solve those problems even if we take their formalizations with auxiliary lemmas and different encodings of the natural numbers. In Table 6 we report problems with valid properties solved by CVC4 with induction, or by AdtInd, or by Eldarica, that are not solved by ADTREM.

<i>Problem</i>	<i>Property</i>
CLAM goal4	$\forall x. \text{len}(\text{append}(x,x)) = 2 \text{len}(x)$
CLAM goal6	$\forall x, y. \text{len}(\text{rev}(\text{append}(x,y))) = \text{len}(x) + \text{len}(y)$
IsaPlanner goal52	$\forall n, l. \text{count}(n,l) = \text{count}(n, \text{rev}(l))$
IsaPlanner goal80	$\forall l. \text{sorted}(\text{sort}(l))$

Table 5. Problems solved by ADTREM and solved by neither CVC4 with induction nor AdtInd nor Eldarica.

<i>Problem</i>	<i>Property</i>	<i>Solved by</i>
CLAM goal18	$\forall x, y. \text{rev}(\text{append}(\text{rev}(x),y)) = \text{append}(\text{rev}(y),x)$	CVC4 with induction
CLAM goal76	$\forall x, y. \text{append}(\text{revflat}(x),y) = \text{qrevflat}(x,y)$	AdtInd
Leon amortize-queue-goal3	$\forall x. \text{len}(\text{qrev}(x)) = \text{len}(x)$	Eldarica

Table 6. Problems solved by CVC4 with induction or AdtInd or Eldarica and not solved by ADTREM.

9 Related Work and Conclusions

This paper is an improved, extended version of a paper that appears in the Proceedings of the 10th International Joint Conference on Automated Reasoning (IJCAR 2020) [18]. The paper was also presented at the 35th Italian Conference on Computational Logic (CILC 2020). Besides detailed proofs and examples, the main, new contribution consists in addressing the problem of the *completeness* of the transformations. In the IJCAR paper, we proved only a *soundness* property, ensuring that if the transformed clauses are satisfiable, then so are the original clauses. In this paper, we identify some sufficient conditions, related to the functionality of the difference predicates introduced by the transformation algorithm \mathcal{R} , which guarantee completeness (that is, the converse of soundness) stating that if the transformed clauses are unsatisfiable, then so are the original clauses. We have also extended our benchmark and our implementation, and we have shown that those sufficient conditions are indeed satisfied in many interesting, non trivial examples. Finally, we have extended the comparison of our experimental results with the ones obtained by the AdtInd solver that extends CHC solving with induction on the ADT structure and lemma generation [54].

Inductive reasoning is supported, with different degrees of human intervention, by many theorem provers, such as ACL2 [34], CLAM [30], Isabelle [41], HipSpec [8], Zeno [48], and PVS [42]. The combination of inductive reasoning and SMT solving techniques has been exploited by many tools for program verification [37, 44, 46, 50, 53, 54].

Leino [37] integrates inductive reasoning into the Dafny program verifier by implementing a simple strategy that rewrites user-defined properties that may benefit from induction into proof obligations to be discharged by Z3 [19]. The advantage of this technique is that it fully decouples inductive reasoning from SMT solving. Hence, no extensions to the SMT solver are required.

In order to extend CVC4 with induction, Reynolds and Kuncak [46] also consider the rewriting of formulas that may take advantage from inductive reasoning, but this is done dynamically, during the proof search. This approach allows CVC4 to perform the rewritings lazily, whenever new formulas are generated during the proof search, and to use the partially solved conjecture for generating lemmas that may help in the proof of the initial conjecture.

The issue of generating suitable lemmas during inductive proofs has been also addressed by Yang et al. [54] and implemented in AdtInd. In order to conjecture new lemmas, their algorithm makes use of a syntax-guided synthesis strategy driven by a grammar, which is dynamically generated from user-provided templates and the function and predicate symbols encountered during the proof search. The derived lemma conjectures are then checked by the SMT solver Z3. In our approach, the introduction of difference predicates can be viewed as the transformational counterpart of lemma generation. When we prove the satisfiability of the transformed CHCs, we also compute models of the difference predicates, which indeed correspond to valid properties. These properties could also be added to the background theory and used as axioms by solvers or theorem provers (see the example in Section 2). The experimental evaluation of Section 8, shows that, on a small but non-trivial benchmark, our tool ADTREM performs better than AdtInd.

In order to take full advantage of the efficiency of SMT solvers in checking satisfiability of quantifier-free formulas over LIA, ADTs, and finite sets, the Leon verification system [50] implements an SMT-based solving algorithm to check the satisfiability of formulas involving recursively defined first-order functions. The algorithm interleaves the unrolling of recursive functions and the SMT solving of the formulas generated by the unrolling. Leon can be used to prove properties of Scala programs with ADTs and integrates with the Scala compiler and the SMT solver Z3. A refined version of that algorithm, restricted to *catamorphisms*, has been implemented into a solver-agnostic tool, called RADA [44].

In the context of CHCs, Unno et al. [53] have proposed a proof system that combines inductive theorem proving with SMT solving. This approach uses Z3 with the PDR engine [28] to discharge proof obligations generated by the proof system, and has been applied to prove relational properties of OCaml programs.

Recent work by Kostyukov et al. [36] proposes a method for proving the satisfiability of CHCs over ADTs by computing models represented by finite

tree automata. The tool based on this approach, called RegInv, is applied to the problem of computing invariants of programs that manipulate ADTs and it is shown to be more practical, in some cases, than state-of-the-art CHC solvers that compute invariants represented by first-order logic formulas. In our approach we do not provide an explicit representation of the model of the initial, non-transformed CHCs, while the transformed clauses have basic types only, and thus we need not extend the usual notion of a model.

The distinctive feature of the technique presented in this paper is that it does not make use of any explicit inductive reasoning, but it follows a transformational approach. First, the problem of verifying the validity of a universally quantified formula over ADTs is reduced to the problem of checking the satisfiability of a set of CHCs. Then, this set of CHCs is transformed with the aim of deriving a set of CHCs over basic types (such as integers and booleans) only, whose satisfiability implies the satisfiability of the original set. In this way, the reasoning on ADTs is separated from the reasoning on satisfiability, which can be performed by specialized engines for CHCs on basic types (e.g., Eldarica [29] and Z3-SPACER [35]). Some of the ideas presented here have been explored in previous work [16, 17], but there neither formal results nor an automated strategy were presented.

A key success factor of our technique is the introduction of difference predicates, which, as already mentioned, can be viewed as a form of automatic lemma generation. Indeed, as shown in Section 8, the use of difference predicates greatly increases the power of CHC solving with respect to previous techniques based on the transformational approach, which do not use difference predicates [15].

As future work, we plan to apply our transformation-based verification technique to more complex program properties, such as relational properties [12, 14]. Another important problem to study is the termination of the ADT removal algorithm \mathcal{R} . As already mentioned, due to the undecidability of the satisfiability problem for CHCs, there is no sound and complete algorithm that always terminates and removes all ADTs. Thus, we can tackle this problem in two ways: (1) by identifying restricted classes of CHCs for which a sound and complete ADT removal algorithm terminates, or (2) by retaining soundness only and designing a suitable generalization strategy that guarantees the introduction of a finite set of new definitions during ADT removal. For Point (2), it would be interesting to explore how to combine the introduction of difference predicates with the most specific generalization techniques used in automated theorem proving [6] and logic program transformation [20, 49].

Acknowledgements

We warmly thank Francesco Calimeri, Simona Perri, and Ester Zumpano for inviting us to submit this improved, extended version of the paper we presented at the 35th Italian Conference on Computational Logic (CILC 2020) held in Rende, Italy, 13–15 October 2020. We also thank the anonymous reviewers for their helpful comments and suggestions.

References

1. R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.
2. K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–576. Elsevier, 1990.
3. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *23rd CAV '11*, Lecture Notes in Computer Science 6806, pages 171–177. Springer, 2011.
4. C. W. Barrett and C. Tinelli. Satisfiability modulo theories. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer, 2018.
5. N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, and W. Schulte, editors, *Fields of Logic and Computation (II)*, Lecture Notes in Computer Science 9300, pages 24–51. Springer, 2015.
6. A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (I)*, pages 845–911. North Holland, 2001.
7. A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In N. Piterman and S. Smolka, editors, *19th TACAS '13*, Lecture Notes in Computer Science 7795, pages 93–107. Springer, 2013.
8. K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In M. P. Bonacina, editor, *CADE-24*, Lecture Notes in Artificial Intelligence 7898, pages 392–406. Springer, 2013.
9. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, Sept. 2003.
10. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th POPL '78*, pages 84–96. ACM, 1978.
11. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A tool for verifying programs through transformations. In *20th TACAS '14*, Lecture Notes in Computer Science 8413, pages 568–574. Springer, 2014.
12. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Relational verification through Horn clause transformation. In X. Rival, editor, *23rd SAS '16*, Lecture Notes in Computer Science 9837, pages 147–169. Springer, 2016.
13. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions via program specialization. *Science of Computer Programming*, 147:78–108, 2017. Selected and Extended papers from the International Symposium on Principles and Practice of Declarative Programming 2015.
14. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Predicate Pairing for program verification. *Theory and Practice of Logic Programming*, 18(2):126–166, 2018.
15. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Solving Horn clauses on inductive data types without induction. *Theory and Practice of Logic Programming*, 18(3-4):452–469, 2018.

16. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Lemma generation for Horn clause satisfiability: A preliminary study. In A. Lisitsa and A. P. Nemytykh, editors, *7th Int. Workshop on Verification and Program Transformation, VPT@Programming*, EPTCS no. 299, pages 4–18, 2019.
17. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Proving properties of sorting programs: A case study in Horn clause verification. In E. De Angelis, G. Fedyukovich, N. Tzevelekos, and M. Ulbrich, editors, *6th HCVS and 3rd PERR*, EPTCS no. 296, pages 48–75, 2019.
18. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Removing algebraic data types from constrained Horn clauses using difference predicates. In N. Peltier and V. Sofronie-Stokkermans, editors, *Proceedings of the International Joint Conference on Automated Reasoning, IJCAR 2020*, Lecture Notes in Artificial Intelligence 12166, pages 83–102. Springer, 2020.
19. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *14th TACAS '08*, Lecture Notes in Computer Science 4963, pages 337–340. Springer, 2008.
20. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming*, 41(2–3):231–277, 1999.
21. L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In F. Baader, editor, *CADE-19*, Lecture Notes in Computer Science 2741, pages 279–283. Springer, 2003.
22. H. Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York, 1972.
23. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
24. F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In K.-K. Lau and M. Bruynooghe, editors, *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, pages 292–340. Springer-Verlag, 2004.
25. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming*, 13(2):175–199, 2013.
26. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *CAV '97*, LNCS 1254, pages 72–83, Berlin, Heidelberg, 1997. Springer.
27. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *33rd ACM SIGPLAN Conf. Programming Language Design and Implementation, PLDI '12*, pages 405–416, 2012.
28. K. Hoder and N. Bjørner. Generalized property directed reachability. In A. Cimatti and R. Sebastiani, editors, *15th SAT '12*, Lecture Notes in Computer Science 7317, pages 157–171. Springer, 2012.
29. H. Hojjat and P. Rümmer. The ELDARICA Horn solver. In N. Bjørner and A. Gurfinkel, editors, *Formal Methods in Computer Aided Design, FMCAD 2018*, pages 1–7. IEEE, 2018.
30. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1):79–111, Mar. 1996.
31. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
32. M. Johansson, L. Dixon, and A. Bundy. Case-analysis for rippling and inductive proof. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science 6172, pages 291–306. Springer, 2010.

33. B. Kafle, J. P. Gallagher, and J. F. Morales. RAHFT: A tool for verifying Horn clauses using abstract interpretation and finite tree automata. In *28th CAV '16, Part I*, Lecture Notes in Computer Science 9779, pages 261–268. Springer, 2016.
34. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
35. A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke. Automatic abstraction in SMT-based unbounded software model checking. In N. Sharygina and H. Veith, editors, *25th CAV '13*, Lecture Notes in Computer Science 8044, pages 846–862. Springer, 2013.
36. Y. Kostyukov, D. Mordvinov, and G. Fedyukovich. Beyond the elementary representations of program invariants over algebraic data types. In S. N. Freund and E. Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Canada, June 20-25, 2021*, pages 451–465. ACM, 2021.
37. K. Leino. Automating induction with an SMT solver. In V. Kuncak and A. Rybalchenko, editors, *13th VMCAI*, Lecture Notes in Computer Science 7148, pages 315–331. Springer, 2012.
38. X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system, Release 4.06. Documentation and user’s manual, Institut National de Recherche en Informatique et en Automatique, France, 2017.
39. M. Leuschel and M. Butler. ProB: A model checker for B. In *FME 2003: Formal Methods*, Lecture Notes in Computer Science 2805, pages 855–874. Springer, 2003.
40. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.
41. T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.
42. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *CADE-11*, pages 748–752. Springer, 1992.
43. A. Pettorossi, M. Proietti, and V. Senni. Constraint-based correctness proofs for logic program transformations. *Formal Aspects of Computing*, 24:569–594, 2012.
44. T.-H. Pham, A. Gacek, and M. W. Whalen. Reasoning about algebraic data types with abstractions. *J. Autom. Reason.*, 57(4):281–318, Dec. 2016.
45. M. O. Rabin. Decidable theories. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 595–629. North-Holland, 1977.
46. A. Reynolds and V. Kuncak. Induction for SMT solvers. In D. D’Souza, A. Lal, and K. G. Larsen, editors, *16th VMCAI*, Lecture Notes in Computer Science 8931, pages 80–98. Springer, 2015.
47. H. Seki. On inductive and coinductive proofs via unfold/fold transformations. In D. De Schreye, editor, *19th LOPSTR '09*, Lecture Notes in Computer Science 6037, pages 82–96. Springer, 2010.
48. W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In C. Flanagan and B. König, editors, *18th TACAS '12*, pages 407–421. Springer, 2012.
49. M. H. Sørensen and R. Glück. An algorithm of generalization in positive super-compilation. In J. W. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium (ILPS '95)*, pages 465–479. MIT Press, 1995.
50. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In E. Yahav, editor, *18th SAS '11*, Lecture Notes in Computer Science 6887, pages 298–315. Springer, 2011.

51. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming, ICLP '84*, pages 127–138, Uppsala, Sweden, 1984. Uppsala University.
52. H. Tamaki and T. Sato. A generalized correctness proof of the unfold/fold logic program transformation. Technical Report 86-4, Ibaraki University, Japan, 1986.
53. H. Unno, S. Torii, and H. Sakamoto. Automating induction for solving Horn clauses. In R. Majumdar and V. Kuncak, editors, *29th CAV '17, Part II*, Lecture Notes in Computer Science 10427, pages 571–591. Springer, 2017.
54. W. Yang, G. Fedyukovich, and A. Gupta. Lemma synthesis for automating induction over algebraic data types. In T. Schiex and S. de Givry, editors, *25th Int. Conf. Principles and Practice of Constraint Programming, CP 2019*, Lecture Notes in Computer Science 11802, pages 600–617. Springer, 2019.