Removing Algebraic Data Types
from Constrained Horn Clauses
Using Difference Predicates*

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

\textsuperscript{1} DEC, University ‘G. d’Annunzio’, Chieti-Pescara, Italy
  fabio.fioravanti@unich.it
\textsuperscript{2} DICII, University of Rome ‘Tor Vergata’, Italy
  pettorossi@info.uniroma2.it
\textsuperscript{3} IASI-CNR, Rome, Italy
  \{emanuele.deangelis,maurizio.proietti\}@iasi.cnr.it

Abstract. We address the problem of proving the satisfiability of Constrained Horn Clauses (CHCs) with Algebraic Data Types (ADTs), such as lists and trees. We propose a new technique for transforming CHCs with ADTs into CHCs where predicates are defined over basic types, such as integers and booleans, only. Thus, our technique avoids the explicit use of inductive proof rules during satisfiability proofs. The main extension over previous techniques for ADT removal is a new transformation rule, called \textit{differential replacement}, which allows us to introduce auxiliary predicates corresponding to the lemmas that are often needed when making inductive proofs. We present an algorithm that uses the new rule, together with the traditional folding/unfolding transformation rules, for the automatic removal of ADTs. We prove that if the set of the transformed clauses is satisfiable, then so is the set of the original clauses. By an experimental evaluation, we show that the use of the differential replacement rule significantly improves the effectiveness of ADT removal, and we show that our transformation-based approach is competitive with respect to a well-established technique that extends the CVC4 solver with induction.

1 Introduction

\textit{Constrained Horn Clauses} (CHCs) constitute a fragment of the first order predicate calculus, where the Horn clause format is extended by allowing \textit{constraints} on specific domains to occur in clause premises. CHCs have gained popularity as a suitable logical formalism for automatic program verification \cite{3}. 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).
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 *background theory* [2]. Recent advances in the field have led to the development of a number of very powerful and sophisticated SMT (and, in particular, CHC) solvers, which aim at solving satisfiability problems with respect to a large variety of background theories. Among SMT solvers, we would like to mention CVC4 [1], MathSAT [5], and Z3 [13], and among solvers with specialized engines for CHC satisfiability, we recall Eldarica [21], HSF [19], RAHFT [24], and Spacer [28].

Even if SMT algorithms for unrestricted first order formulas suffer from well-known incompleteness limitations due to general undecidability results, most of the above mentioned tools work well in practice when acting on background (or 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 *algebraic data types* (ADTs) which are inductively defined, such as lists and trees, then the SMT/CHC solvers usually show a very poor performance, because they do not incorporate induction principles relative to the ADT in use.

To mitigate this difficulty, some SMT/CHC solvers have been enhanced by incorporating appropriate induction principles [38, 43, 44], similarly to what has been done in automated theorem provers [4]. 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 [10], 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 (at transformation time) from the concern of dealing with simpler, non-inductive constraint theories (at solving time), thus avoiding the complex interaction between inductive reasoning and constraint solving. It has been shown [10] that the transformational approach compares well with induction-based tools in the case where 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.

In this paper we will extend the transformational approach by introducing a transformation rule that corresponds to lemma generation when making inductive proofs. Our main contributions are as follows.

1. We present a new transformation rule, called *differential replacement*, based on the introduction of suitable *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 for CHCs [16] 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.

2. We develop a transformation algorithm that removes ADTs from CHCs by
applying the fold/unfold and the differential replacement rules in a fully automated way.

(3) Due to the undecidability of the satisfiability problem for CHCs, our technique for ADT removal is incomplete. 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 [38].

The paper is structured as follows. In Section 2 we 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 their soundness. In Section 5 we present the transformation algorithm that uses the rules introduced in Section 4 for removing ADTs from sets of CHCs. In Section 6 we illustrate the AdtRem tool and we present the experimental results we have obtained. Finally, in Section 7 we discuss the related work and we make some concluding considerations.

2 A Motivating Example

Let us consider the following functional program Reverse, which we write using the OCaml syntax [30]:

```ocaml
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 rev l = match l with
  | Nil   -> Nil  | Cons(x, xs) -> append (rev xs) (Cons(x, Nil));;
let rec len l = match l with
  | Nil   -> 0    | Cons(x, xs) -> 1 + len xs;;
```

The functions `append`, `rev`, and `len` compute list concatenation, list reversal, and list length, respectively. Suppose we want to prove the following property:

\[ \forall x, y, z. \text{len} (\text{rev} (\text{append} x y z)) = (\text{len} x) + (\text{len} y) \quad (1) \]

Inductive theorem provers construct a proof of this property by induction on the structure of the list `l`, by assuming the knowledge of the following lemma:

\[ \forall x, l. \text{len} (\text{append} l (\text{Cons}(x,\text{Nil}))) = (\text{len} l) + 1 \quad (2) \]

The approach we follow in this paper avoids the explicit use of induction principles and also the knowledge of ad hoc lemmas. First, we consider the translation of Property (1) into a set of constrained Horn clauses [10, 13], as follows: 

4 In the examples, we use Prolog syntax for writing clauses, instead of the more verbose SMT-LIB syntax. The predicates `\neq` (different from), `=` (equal to), `<` (less-than), `\geq` (greater-than-or-equal-to) denote constraints between integers. The last argument of a Prolog predicate stores the value of the corresponding function.
The set of clauses 1–7 is satisfiable if and only if Property (1) holds. However, state-of-the-art CHC solvers, such as Z3 or Eldarica, fail to prove the satisfiability of clauses 1–7, because those solvers do not incorporate any induction principle on lists. Moreover, some tools that extend SMT solvers with induction [38, 43] fail on this particular example because they are not able to introduce Lemma (2).

To overcome this difficulty, we apply the transformational approach based on the fold/unfold rules [10], whose objective is to transform a given set of clauses into a new set without occurrences of list variables, whose satisfiability can be checked by using CHC solvers based on the theory of Linear Integer Arithmetic only. The soundness of the transformation rules ensures that the satisfiability of the transformed clauses implies the satisfiability of the original ones. We apply the Elimination Algorithm [10] as follows. First, we introduce a new clause:

1. new1(N0,N1,N2) :- append(Xs,Ys,Zs), rev(Zs,Rs), len(Xs,N0), len(Ys,N1), len(Rs,N2).

whose body is made out of the atoms of clause 1 which have at least one list variable, and whose head arguments are the integer variables of the body. By folding, from clause 1 we derive a new clause without occurrences of lists:

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

We would like to fold clause 11 using clause 8, so as to derive a recursive definition of new1 without lists. Unfortunately, this folding step cannot be performed because the body of clause 11 does not contain a variant of the body of clause 8, and hence the Elimination Algorithm fails to eliminate lists in this example.

Thus, now we depart from the Elimination Algorithm and we continue our derivation by observing that the body of clause 11 contains the subconjunction ‘append(Xs,Ys,Zs), rev(Zs,Rs), len(Xs,N0), len(Ys,N1)’ of the body of clause 8. Then, in order to find a variant of the whole body of clause 8, we may replace in clause 11 the remaining subconjunction ‘append(Rs,[X],R1s), len(R1s,N21)’ by the new subconjunction ‘len(Rs,N2), diff(N2,X,N21)’, where diff is a predicate, called difference predicate, defined as follows:

3. diff(N2,X,N21) :- append(Rs,[X],R1s), len(R1s,N21), len(Rs,N2).
Now, we can fold clause 13 using clause 8 and we derive a new clause without list arguments:

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

At this point, we are left with the task of removing list arguments from clauses 10 and 12. As the reader may verify, this can be done by applying the Elimination Algorithm without the need of introducing additional difference predicates. By doing so, we get the following final set of clauses without list arguments:

false :- N2\=N0+N1, new1(N0,N1,N2).
new1(N0,N1,N2) :- N0=0, new2(N1,N2).
new1(N0,N1,N2) :- N0=N+1, new1(N,N1,M), diff(M,X,N2).
new2(M1,N1) :- M1=M+1, new2(M,N), diff(N,X,N1).
diff(N,X,M) :- M=N+1, N>=0.

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

The Eldarica CHC solver proves the satisfiability of this set of clauses by computing the following model (here we use a Prolog-like syntax):

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

Finally, we note that if in clause 12 we substitute the atom diff(N2,X,N21) by its model computed by Eldarica, namely the constraint ’N21=N2+1, N2>=0’, we get exactly the CHC translation of Lemma (2). Thus, in some cases, the introduction of the difference predicates can be viewed as a way of automatically introducing the lemmas needed when performing inductive proofs.

3 Constrained Horn Clauses

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 \(C\) denote the set of all constraints. Let \(L\) be a typed first order language with equality \([15]\) which includes the language of \(LIA \cup Bool\). Let \(Pred\) be a set of predicate symbols in \(L\) not occurring in the language of \(LIA \cup Bool\).

The integer and boolean types are said to be the basic types. For reasons of simplicity we do not consider any other basic types, such as real number, arrays, and bit-vectors, which are usually supported by SMT solvers \([1, 13, 21]\). We assume that all non-basic types are specified by suitable data-type declarations (such as the \texttt{declare-datatypes} declarations adopted by SMT solvers), and are collectively called algebraic data types (ADTs).

An atom is a formula of the form \(p(t_1, \ldots, t_m)\), where \(p\) is a typed predicate symbol in \(Pred\), and \(t_1, \ldots, t_m\) are typed terms constructed out of individual variables, individual constants, and function symbols. A constrained Horn clause (or simply, a clause, or a CHC) is an implication of the form \(A \leftarrow c, B\) (for clauses we use the logic programming notation, where comma denotes conjunction). The conclusion (or head) \(A\) is either an atom or \(false\), the premise (or body) is the
conjunction of a constraint $c \in C$, and a (possibly empty) conjunction $B$ of atoms. If $A$ is an atom of the form $p(t_1, \ldots, t_n)$, 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 all variables in a clause are universally quantified in front, and thus we can freely rename those variables. 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 $\text{vars}(t)$ we denote the set of all variables occurring in $t$. Similarly for the set of all free variables occurring in a formula. Given a formula $\varphi$ in $\mathcal{L}$, we denote by $\forall(\varphi)$ its universal closure.

Let $\mathbb{D}$ be the usual interpretation for the symbols in $\text{LIA} \cup \text{Bool}$, and let a $\mathbb{D}$-interpretation be an interpretation of $\mathcal{L}$ that, for all symbols occurring in $\text{LIA} \cup \text{Bool}$, agrees with $\mathbb{D}$.

A set $P$ of CHCs is satisfiable if it has a $\mathbb{D}$-model and it is unsatisfiable, otherwise. Given two $\mathbb{D}$-interpretations $I$ and $J$, we say that $I$ is included in $J$ if for all ground atoms $A$, $I \models A$ implies $J \models A$. Every set $P$ of definite clauses is satisfiable and has a least (with respect to set inclusion) $\mathbb{D}$-model, denoted $M(P)$.

Thus, if $P$ is any set of constrained Horn clauses and $Q$ is the set of the goals in $P$, then we define $\text{Definite}(P) = P \setminus Q$. We have that $P$ is satisfiable if and only if $M(\text{Definite}(P)) \models Q$.

In our notations, we will often use tuples of variables as arguments of predicates and we write $p(X, Y)$, instead of $p(X_1, \ldots, X_m, Y_1, \ldots, Y_n)$, whenever the specific values of $m$ ($\geq 0$) and $n$ ($\geq 0$) are not relevant. Whenever the order of the variables is not relevant, we will feel free to identify finite tuples of distinct variables with finite sets, and we will extend to finite tuples the usual operations and relations on finite sets. Given two tuples $X$ and $Y$ of distinct elements, (i) their union $X \cup Y$ is obtained by concatenating them and removing all duplicated occurrences of elements, (ii) their intersection $X \cap Y$ is obtained by removing from $X$ the elements which do not occur in $Y$, (iii) their difference $X \setminus Y$ is obtained by removing from $X$ the elements which occur in $Y$, and (iv) $X \subseteq Y$ holds if every element of $X$ occurs in $Y$. The equality between two $m$-tuples is defined as follows: for all $m \geq 0$, $(u_1, \ldots, u_m) = (v_1, \ldots, v_m)$ 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 $\text{vars}(A) = X \cup Y$. Let $P$ be a set of definite clauses. We say that the atom $A(X, Y)$ is functional from $X$ to $Y$ with respect to $P$ if

(F1) $M(P) \models \forall X, Y, Z. A(X, Y) \wedge A(X, Z) \rightarrow Y = Z$

The reference to the set $P$ of definite clauses is omitted, when understood from the context. Given a functional atom $A(X, Y)$, we say that $X$ and $Y$ are its input and output (tuples of) variables, respectively. The atom $A(X, Y)$ is said to be total from $X$ to $Y$ with respect to $P$ if

(F2) $M(P) \models \forall X \exists Y. A(X, Y)$

If $A(X, Y)$ is a total, functional atom from $X$ to $Y$, we may write $A(X; Y)$, instead of $A(X, Y)$. For instance, $\text{append}(X_s, Y_s, Z_s)$ is a total, functional atom from $(X_s, Y_s)$ to $Z_s$ with respect to the set of clauses 1–7 of Section 2.
Now we extend the above notions from atoms to conjunctions of atoms. Let $F$ be a conjunction $A_1(X_1; Y_1), \ldots, A_n(X_n; Y_n)$ such that: (i) $X = (\bigcup_{i=1}^{n} X_i) \setminus (\bigcup_{i=1}^{n} Y_i)$, (ii) $Y = (\bigcup_{i=1}^{n} Y_i)$, and (iii) for $i = 1, \ldots, n$, $Y_i$ is disjoint from $(\bigcup_{j=1}^{i} X_j) \cup (\bigcup_{j=1}^{i-1} Y_j)$. Then, the conjunction $F$ is said to be a total, functional conjunction from $X$ to $Y$ and it is also written as $F(X; Y)$. For $F(X; Y)$, the above properties (F1) and (F2) hold if we replace $A$ by $F$. For instance, $\text{append}(Xs, Ys, Zs)$, $\text{rev}(Zs, Rs)$ is a total, functional conjunction from $(Xs, Ys)$ to $(Zs, Rs)$ with respect to the set of clauses 1–7 of Section 2.

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 their soundness. We refer to Section 2 for examples of how the rules are applied.

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 $\ell : \text{Pred} \rightarrow \mathbb{N}$. 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 $H \leftarrow c, A_1, \ldots, A_n$ is stratified with respect to $\ell$ if, for $i = 1, \ldots, n$, $\ell(H) \geq \ell(A_i)$. A set $P$ of CHCs is stratified w.r.t. $\ell$ if all clauses in $P$ are stratified w.r.t. $\ell$. Clearly, for every set $P$ of CHCs, there exists a level mapping $\ell$ such that $P$ is stratified w.r.t. $\ell$.

A transformation sequence from $P_0$ to $P_n$ is a sequence $P_0 \Rightarrow P_1 \Rightarrow \ldots \Rightarrow P_n$ of sets of CHCs such that, for $i = 0, \ldots, n-1$, $P_{i+1}$ is derived from $P_i$, denoted $P_i \Rightarrow P_{i+1}$, by applying one of the following Rules R1–R7. We assume that $P_0$ is stratified w.r.t. a given level mapping $\ell$.

(R1) Definition Rule. Let $D$ be the clause $\text{newp}(X_1, \ldots, X_k) \leftarrow c, A_1, \ldots, A_n$, where: (i) $\text{newp}$ is a predicate symbol in $\text{Pred}$ not occurring in the sequence $P_0 \Rightarrow P_1 \Rightarrow \ldots \Rightarrow P_i$, (ii) $c$ is a constraint, (iii) the predicate symbols of $A_1, \ldots, A_n$ occur in $P_0$, and (iv) $(X_1, \ldots, X_k) \subseteq \text{vars}(c, A_1, \ldots, A_n)$. Then, $P_{i+1} = P_i \cup \{D\}$.

We set $\ell(\text{newp}) = \max \{\ell(A_i) \mid i = 1, \ldots, n\}$.

For $i = 0, \ldots, n$, by $\text{Defs}_i$ we denote the set of clauses, called definitions, introduced by Rule R1 during the construction of $P_0 \Rightarrow P_1 \Rightarrow \ldots \Rightarrow P_i$. Thus, $\emptyset = \text{Defs}_0 \subseteq \text{Defs}_1 \subseteq \ldots$ However, by using Rules R2–R7 we can replace a definition in $P_i$, for $i > 0$, and hence it may happen that $\text{Defs}_{i+1} \not\subseteq P_{i+1}$.

(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 $\text{vars}(C) \cap \text{vars}(P_0) = \emptyset$. Let $\text{Cls}$: $\{K_1 \leftarrow c_1, B_1, \ldots, K_m \leftarrow c_m, B_m\}$, $m \geq 0$, be the set of clauses in $P_0$, such that: (1) for $j = 1, \ldots, m$, (1) there exists a most general unifier $\vartheta_j$ of $A$ and $K_j$, and (2) the conjunction of constraints $(c, c_j)\vartheta_j$ is satisfiable. Let $\text{Unf}(C, A, P_0) = \{(H \leftarrow c, c_j, G_L, B_j, G_R)\vartheta_j \mid j = 1, \ldots, m\}$. Then, by unfolding $C$ w.r.t. $A$, we derive the set $\text{Unf}(C, A, P_0)$ of clauses and we get $P_{i+1} = (P_i \setminus \{C\}) \cup \text{Unf}(C, A, P_0)$. 


When we apply Rule R2, we say that, for $j = 1, \ldots, m$, 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 $C$.

(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 clause in $\text{Defs}_i$. Suppose that: (i) either $H$ is false or $\ell(H) \geq \ell(K)$, and (ii) there exists a substitution $\vartheta$ such that $Q = B \vartheta$ and $\mathcal{D} \models \forall(c \rightarrow d \vartheta)$. By folding $C$ using definition $D$, we derive clause $E: H \leftarrow c, G_L, K \vartheta, G_R$, and we get $P_{i+1} = (P_i \setminus \{C\}) \cup \{E\}$.

(R4) **Clause Deletion Rule.** Let $C: H \leftarrow c, G$ be a clause in $P_i$ such that the constraint $c$ is unsatisfiable. Then, we get $P_{i+1} = P_i \setminus \{C\}$.

(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 total, functional conjunction in $\text{Definite}(P_0) \cup \text{Defs}_i$. By functionality, from $C$ we derive clause $D: H \leftarrow c, Y = Z, G_L, F(X; Y), G_R$, and we get $P_{i+1} = (P_i \setminus \{C\}) \cup \{D\}$.

(R6) **Totality Rule.** Let $C: H \leftarrow c, G_L, F(X; Y), G_R$ be a clause in $P_i$ such that $Y \cap \text{vars}(H \leftarrow c, G_L, G_R) = \emptyset$ and $F(X; Y)$ is a total, functional conjunction in $\text{Definite}(P_0) \cup \text{Defs}_i$. By totality, from $C$ we derive clause $D: H \leftarrow c, G_L, G_R$ and we get $P_{i+1} = (P_i \setminus \{C\}) \cup \{D\}$.

In practice, we do not need to prove the functionality and totality properties of the conjunction $F(X; Y)$ of atoms each time we apply Rules R5 and R6. Indeed, when we apply those rules, we will only use properties of atoms that directly derive from the fact that the initial set of clauses is the result of translating a given terminating functional program.

(R7) **Differential Replacement Rule.** Let $C: H \leftarrow c, G_L, F(X; Y), G_R$ be a clause in $P_i$, and let $D: \text{diff}(Z) \leftarrow d, F(X; Y), R(V; W)$ be a definition clause in $\text{Defs}_i$, where: (i) $F(X; Y)$ and $R(V; W)$ are total, functional conjunctions with respect to $\text{Definite}(P_0) \cup \text{Defs}_i$, (ii) $W \cap \text{vars}(C) = \emptyset$, (iii) $\mathcal{D} \models \forall(c \rightarrow d)$, and (iv) $\ell(H) > \ell(\text{diff})$. By differential replacement, we derive $E: H \leftarrow c, G_L, R(V; W), \text{diff}(Z), G_R$ and we get $P_{i+1} = (P_i \setminus \{C\}) \cup \{E\}$.

Rule R7 has a very general form, and in fact in the transformation algorithm of Section 5 we will use a specific instance which is sufficient for ADT removal (see, in particular, the Diff-Introduce step). However, the general formulation eases the proof of the Soundness Theorem below, which extends to Rules R1–R7 correctness results relative to transformations of logic and constraint logic programs [16, 17, 39, 42] (see the Appendix for a proof).

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

(U) for $i = 1, \ldots, n-1$, if $P_i \Rightarrow P_{i+1}$ by folding a clause in $P_i$ using a definition $D: H \leftarrow c, B$ in $\text{Defs}_i$, then, for some $j \in \{1, \ldots, i-1, i+1, \ldots, n-1\}$, $P_j \Rightarrow P_{j+1}$ 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 of clauses, it suffices: (i) to construct a transformation sequence \( P_0 \Rightarrow \ldots \Rightarrow P_n \), and then (ii) to prove that \( P_n \) is satisfiable. Note, however, that if Rule R7 is used, it may happen that \( P_0 \) is satisfiable and \( P_n \) is unsatisfiable, that is, some false counterexamples to satisfiability, so-called false positives, may be generated, as we now show.

**Example 1.** Let us consider the following set \( P_1 \) of clauses derived by adding the definition clause \( D \) to the initial set \( P_0 = \{ C, 1, 2, 3 \} \) of clauses:

\[
\begin{align*}
C. \ false & :- X=0, Y>0, a(X,Y). \\
1. \ a(X,Y) & :- X=<0, Y=0. \\
2. \ a(X,Y) & :- X>0, Y=1. \\
3. \ r(X,W) & :- W=1. \\
D. \ diff(Y,W) & :- a(X,Y), r(X,W).
\end{align*}
\]

where: (i) \( a(X,Y) \) is a total, functional atom from \( X \) to \( Y \), (ii) \( r(X,W) \) is a total, functional atom from \( X \) to \( W \), and (iii) \( D \) is a definition in \( \text{Defs}_1 \). By applying Rule R7, from \( P_1 \) we derive the set \( P_2 = \{ E, 1, 2, 3, D \} \) of clauses where:

\[
\begin{align*}
E. \ false & :- X=0, Y>0, r(X,W), \text{diff}(Y,W).
\end{align*}
\]

Now we have that \( P_0 \) is satisfiable, while \( P_2 \) is unsatisfiable.

## 5 An Algorithm for ADT Removal

Now we present Algorithm \( R \) for eliminating ADT terms from CHCs, by using the transformation rules presented in Section 4. Algorithm \( R \) automatically introduces suitable difference predicates. If \( R \) terminates, it transforms a set \( \text{Cls} \) of clauses into a new set \( \text{TransfCls} \) where the arguments of all predicates have basic type. Theorem 4 guarantees that if \( \text{TransfCls} \) is satisfiable, then also \( \text{Cls} \) is satisfiable.

Algorithm \( R \) (see Figure 1) removes ADT terms starting from the set \( Gs \) of goals in \( \text{Cls} \). \( R \) collects these goals in \( \text{InCls} \) and stores in \( \text{Defs} \) the definitions of new predicates introduced by Rule R1.

---

**Algorithm \( R \).**

**Input:** A set \( \text{Cls} \) of clauses;  
**Output:** A set \( \text{TransfCls} \) of clauses that have basic types.

Let \( \text{Cls} = Ds \cup Gs \), where \( Ds \) is a set of definite clauses and \( Gs \) is a set of goals;  
\( \text{InCls} := Gs; \text{Defs} := \emptyset; \text{TransfCls} := \emptyset; \)

**while** \( \text{InCls} \neq \emptyset \) **do**

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

Fig. 1. The ADT Removal Algorithm \( R \).
have a basic type. An atom (or clause) has ADTs if at least one of its arguments (or atoms, respectively) has an ADT type.

Given a set (or a conjunction) $S$ of atoms, $\text{SharingBlocks}(S)$ denotes the partition of $S$ with respect to the reflexive, transitive closure $\downarrow_S$ of the relation $\downarrow$ defined as follows. Given two atoms $A_1$ and $A_2$ in $S$, $A_1 \downarrow A_2$ holds if $\text{adt-vars}(A_1) \cap \text{adt-vars}(A_2) \neq \emptyset$. The elements of the partition are called the sharing blocks of $S$.

A generalization of a pair $(c_1, c_2)$ of constraints is a constraint $\alpha(c_1, c_2)$ such that $\mathbb{D} \models \forall(c_1 \rightarrow \alpha(c_1, c_2))$ and $\mathbb{D} \models \forall(c_2 \rightarrow \alpha(c_1, c_2))$ \cite{18}. In particular, we consider the following generalization operator based on widening \cite{7}. Suppose that $c_1$ is the conjunction $(a_1, \ldots, a_m)$ of atomic constraints, then $\alpha(c_1, c_2)$ is defined as the conjunction of all $a_i$'s in $(a_1, \ldots, a_m)$ such that $\mathbb{D} \models \forall(c_2 \rightarrow a_i)$. For any constraint $c$ and tuple of variables $V$, 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 from $\exists Y.c$, where $Y = \text{vars}(c) \setminus V$, by a quantifier elimination algorithm in the theory of booleans and rational numbers. This implementation is safe in our context, and avoids relying on modular arithmetic, as is often done when eliminating quantifiers in LIA \cite{37}.

For two conjunctions $G_1$ and $G_2$ of atoms, $G_1 \subseteq G_2$ holds if $G_1 = (A_1, \ldots, A_n)$ and there exists a subconjunction $(B_1, \ldots, B_n)$ of $G_2$ (modulo reordering) such that, for $i = 1, \ldots, n$, $B_i$ is an instance of $A_i$. A conjunction $G$ of atoms is connected if it consists of a single sharing block.

**Procedure Diff-Define-Fold** (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. This is done by possibly introducing some new definitions (using 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).** We remove the ADT arguments occurring in $B$ by folding $C$ using a definition $D$ introduced at a previous step. Indeed, the head of each definition introduced by Algorithm $R$ is by construction a tuple of variables of basic type.
- **(Generalize).** We introduce a new definition $GenD: \text{genp}(V) \leftarrow \alpha(d, c), B$ whose constraint is obtained by generalizing $(d, c)$, where $d$ is the constraint occurring in an already available definition whose body is $B$. Then, we remove the ADT arguments occurring in $B$ by folding $C$ using $GenD$.
- **(Diff-Introduce).** Suppose that $B$ partially matches the body of an available definition $D: \text{newp}(U) \leftarrow d, B'$, that is, for some substitution $\theta$, $B = (M, F(X; Y)), B' \theta = (M, R(V; W))$. Then, we introduce a difference predicate through the new definition $D: \text{diff}(Z) \leftarrow \pi(c, X), F(X; Y), R(V; W)$, where $Z = \text{bvars}(F(X; Y), R(V; W))$ and, by Rule R7, we replace the conjunction $F(X; Y)$ by $(R(V; W), \text{diff}(Z))$ in the body of $C$, thereby deriving $C'$. Finally, we remove the ADT arguments in $B$ by folding $C'$ using either $D$ or a clause $GenD$ whose constraint is a generalization of the pair $(\theta, c)$ of constraints.

The example of Section 2 allows us to illustrate this (Diff-Introduce) case. With reference to that example, clause $C: H \leftarrow c, G$ that we want to fold
is clause 11, whose body has the single sharing block $B$: \( \text{append}(Xs, Ys, Zs), \text{rev}(Zs, Rs), \text{len}(Xs, N0), \text{len}(Ys, N1), \text{append}(Rs, [X], RIs), \text{len}(RIs, N21) \). Block $B$ partially matches the body \( \text{append}(Xs, Ys, Zs), \text{rev}(Zs, Rs), \text{len}(Xs, N0), \text{len}(Ys, N1), \text{len}(Rs, N2) \) of clause 8 in Section 2 which plays the role of definition $D: \text{newp}(U) \leftarrow d, B'$ in this example. Indeed, we have that:

\[
M = (\text{append}(Xs, Ys, Zs), \text{rev}(Zs, Rs), \text{len}(Xs, N0), \text{len}(Ys, N1)),
\]

\[
F(X; Y) = (\text{append}(Rs, [X], RIs), \text{len}(RIs, N21)), \text{where} \ X = (Rs, X), \ Y = (RIs, N21),
\]
$R(V; W) = \text{len}(Rs, N2)$, where $V = Rs$, $Y = N2$.

In this example, $\vartheta$ is the identity substitution. The condition on the level mapping $\ell$ required in the Diff-Define-Fold Procedure of Figure 2 can be fulfilled by stipulating that $\ell(\text{new}) > \ell(\text{append})$ and $\ell(\text{new}) > \ell(\text{len})$. Thus, the definition $\bar{D}$ to be introduced is:

12. $\text{diff}(N2, X, N21) :- \text{append}(Rs, [X], \text{R1s}), \text{len}(\text{R1s}, N21), \text{len}(Rs, N2)$.

Indeed, we have that: (i) the projection $\pi(c, X)$ is $\pi(N0=0+1, (Rs, X))$, that is, the empty conjunction, (ii) $F(X; Y)$, $R(V; W)$ is the body of clause 12, and (iii) the head variables $N2, X, Y$ are the integer variables in that body. Then, by applying Rule R7, we replace in clause 11 the conjunction ‘$\text{append}(Rs, [X], \text{R1s}), \text{len}(\text{R1s}, N21)$’ by the new conjunction ‘$\text{len}(Rs, N2), \text{diff}(N2, X, N21)$’, hence deriving clause $C'$, which is clause 13 of Section 3. Finally, by folding clause 13 using clause 8, we derive clause 14 of Section 2 which has no list arguments.

- (Project). If none of the previous three cases apply, then we introduce a new definition $\text{ProjC}: \text{newp}(V) \leftarrow \pi(c, V), B$, where $V = \text{bvars}(B)$. Then, we remove the ADT arguments occurring in $B$ by folding $C$ using $\text{ProjC}$.

The Diff-Define-Fold procedure may introduce new definitions with ADTs in their bodies, which are added to the NewDefs and processed by the Unfold procedure. In order to present this procedure, we need the following notions.

The application of Rule R2 is controlled by marking some atoms in the body of a clause as unfoldable. If we unfold w.r.t. atom $A$ clause $C: H \leftarrow c, L, A, R$ the marking of the clauses in Unf($C, A, \text{Ds}$) is handled as follows: the atoms derived from $A$ are not marked as unfoldable and each atom $A'$ inherited from an atom $A$ in the body of $C$ is marked as unfoldable if $A'$ is a source atom.

An atom $A(X; Y)$ in a conjunction $F(V; Z)$ of atoms is said to be a source atom if $X \subseteq V$. Thus, a source atom corresponds to an innermost function call in a given functional expression. For instance, in clause 1 of Section 2 the source atoms are $\text{append}(Xs, Ys, Zs), \text{len}(Xs, N0)$, and $\text{len}(Ys, N1)$. Indeed, the body of clause 1 corresponds to $\text{len}(\text{rev}(\text{append} \ Xs \ Ys)) \neq (\text{len} \ Xs) + (\text{len} \ Ys)$.

An atom $A(X; Y)$ in the body of clause $C: H \leftarrow c, L, A(X; Y), R$ is a head-instance w.r.t. a set $\text{Ds}$ of clauses if, for every clause $K \leftarrow d, B$ in $\text{Ds}$ such that: (1) there exists a most general unifier $\vartheta$ of $A(X; Y)$ and $K$, and (2) the constraint $(c, d)\vartheta$ is satisfiable, we have that $X\vartheta = X$. Thus, the input variables of $A(X; Y)$ are not instantiated by unification. For instance, the atom $\text{append}([X|Xs], Ys, Zs)$ is a head-instance, while $\text{append}(Xs, Ys, Zs)$ is not.

In a set $\text{Cls}$ of clauses, predicate $p$ immediately depends on predicate $q$, if in $\text{Cls}$ there is a clause of the form $p(...)$ $\leftarrow$ $\ldots q(...)$, $\ldots$. The depends on relation is the transitive closure of the immediately depends on relation. Let $\prec$ be a well-founded ordering on tuples of terms such that, for all terms $t, u$, if $t \prec u$, then, for all substitutions $\vartheta, \vartheta \prec u\vartheta$. A predicate $p$ is descending w.r.t. $\prec$ if, for all clauses, $p_t(t; u) \leftarrow c, p_i(t_i; u_i), \ldots, p_n(t_n; u_n)$, for $i = 1, \ldots, n$, 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: $(x_1, \ldots, x_k) \prec (y_1, \ldots, y_m)$ if every $x_i$ is a subterm of some $y_j$ and there exists $x_i$ which is a strict subterm of some $y_j$. For instance, the predicates $\text{append}, \text{rev}$, and $\text{len}$ in the example of Section 2 are all descending.

- Procedure Unfold (see Figure 3) repeatedly applies Rule R2 in two phases. In Phase 1, the procedure unfolds the clauses in $\text{NewDefs}$ w.r.t. at least one source atom. Then, in Phase 2, clauses are unfolded w.r.t. head-instance atoms. Unfolding is repeated only
Removing ADTs from CHCs using Difference Predicates

The termination of the Unfold procedure is ensured by the fact that the unfolding w.r.t. a non-descending atom is done at most once in each phase.

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

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

*Output:* A set UnfCls of clauses.

UnfCls := NewDefs; Mark as unfoldable a nonempty set of source atoms in the body of each clause of UnfCls;

- while there exists a clause $C: H \leftarrow c, L, A, R$ in UnfCls, for some conjunctions $L$ and $R$, such that $A$ is an unfoldable atom do

  UnfCls := $(UnfCls - \{C\}) \cup Unf(C, A, Ds)$;

- Mark as unfoldable all atoms in the body of each clause in UnfCls;

- while there exists a clause $C: H \leftarrow c, L, A, R$ in UnfCls, for some conjunctions $L$ and $R$, such that $A$ is a head-instance atom w.r.t. Ds and $A$ is either unfoldable or descending do

  UnfCls := $(UnfCls - \{C\}) \cup Unf(C, A, Ds)$;

**Fig. 3.** The Unfold Procedure.

- Procedure Replace simplifies some clauses by applying Rules R5 and R6 as long as possible. Replace terminates because each application of either rule decreases the number of atoms.

  Thus, each execution of the Diff-Define-Fold, Unfold, and Replace procedures terminates. However, Algorithm $R$ might not terminate because new predicates may be introduced by Diff-Define-Fold at each iteration of the while-do of $R$. The soundness of $R$ is a consequence of the soundness of the transformation rules (see Appendix).

**Theorem 2 (Soundness of Algorithm $R$).** Suppose that Algorithm $R$ 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.

Algorithm $R$ is not complete, in the sense that, even if Cls is a satisfiable set of input clauses, then $R$ may not terminate or, due to the use of Rule R7, it may terminate with an output set TransfCls of unsatisfiable clauses (see Example 4 in Section 4). However, due to well-known undecidability results for the satisfiability problem of CHCs, this limitation cannot be overcome, unless we restrict the class of clauses that we consider. Studying such restricted classes of clauses is beyond the scope of the present paper and, instead, in the next section, we evaluate the effectiveness of Algorithm $R$ from an experimental point of view.

**6 Experimental Evaluation**

In this section we present the results of some experiments we have performed for assessing the effectiveness of our transformation-based CHC solving technique. We compare our technique with the one proposed by Reynolds and Kuncak [38], which extends the SMT solver CHC4 with inductive reasoning.
Implementation. We have developed the AdtRem tool for ADT removal, which is based on an implementation of Algorithm R in the VeriMAP system [8].

Benchmark suite and experiments. The benchmark suite we have considered, consists of 169 verification problems over inductively defined data structures, such as lists, queues, heaps, and trees, which have been adapted from the benchmark suite considered by Reynolds and Kuncak [38]. These verification problems come from benchmarks used by various automated theorem provers and satisfiability checkers, and in particular: (i) 53 problems from CLAM [22], (ii) 11 from HipSpec [6], (iii) 63 from IsaPlanner [14, 24], and (iv) 42 from Leon [41]. We have performed the following experiments, whose results are summarized in Table 1.

(1) We have considered Reynolds and Kuncak’s dtt encoding of the verification problems, where natural numbers are represented using the built-in SMT type Int, and we have discarded: (i) problems that do not use ADTs, and (ii) problems that cannot be directly represented in Horn clause format. Since AdtRem does not support higher order functions, nor user-provided lemmas, in order to make a comparison between the two approaches on a level playing field, we have replaced higher order functions by suitable first order instances and we have removed all auxiliary lemmas from the input verification problems. We have also replaced the basic functions recursively defined over natural numbers, such as the functions plus and less-or-equal, by LIA constraints, so that the underlying solver is able to use the LIA theory.

(2) We have translated each verification problem into a set, call it P, of CHCs in the Prolog-like syntax supported by AdtRem by using a modified version of the SMT-LIB parser of the ProB system [31]. Then, we have run Eldarica and Z3 which use no induction-based mechanism for handling ADTs, to check the satisfiability of P. Rows ‘Eldarica’ and ‘Z3’ show the number of solved problems.

(3) We have run the ADT removal algorithm R on P to produce a set T of CHCs without ADTs. Row ‘R’ reports the number of problems where the algorithm terminates.

(4) We have converted T into the SMT-LIB format, and then run Eldarica and Z3 for checking its satisfiability. Rows ‘Eldarica\textsubscript{noADT}’ and ‘Z3\textsubscript{noADT}’ report the number of solved problems using Eldarica and Z3, respectively. There was only one false positive (that is, a satisfiable set of clauses transformed into an unsatisfiable one), which we have considered to be an unsolved problem.

(5) In order to assess the improvements due to the differential replacement technique we have applied a modified version of the ADT removal algorithm R on P that does not introduce difference predicates (the Diff-Introduce case of the Diff-Define-Fold Procedure of Figure 2 is never executed). The number of problems where the modified algorithm terminates and the number of solved problems using Eldarica and Z3 are shown between parentheses in rows ‘R’, ‘Eldarica\textsubscript{noADT}’ and ‘Z3\textsubscript{noADT}’, respectively.

(6) We have run the cvc4+ig configuration of the CVC4 solver extended with inductive reasoning [33] on the 169 problems in SMT-LIB format obtained at Step (1). Row ‘CVC4+Ind’ reports the number of solved problems.

Evaluation of Results. The results of our experiments show that ADT removal considerably increases the effectiveness of CHC solvers without inductive reasoning support. For instance, Eldarica is able to solve 15 problems out of the 169 input verification problems, while it solves 122 problems after the removal of ADTs. When using Z3, the improvement is slightly lower, but still very considerable. Note also that, when the

---

5 The tool and the benchmarks are available at https://fmlab.unich.it/adtrem/.

6 More specifically, Eldarica v2.0.1 and Z3 v4.8.0 with the Spacer engine [27].
ADT removal terminates (129 problems out of 169), the solvers are very effective (95% successful verifications for Eldarica). The improvements specifically due to difference predicates can be gauged by observing that, when those predicates are introduced, the number of problems where the ADT removal strategy terminates increases from 94 to 129 and the number of problems solved with Eldarica increases from 94 to 122.

ADTREM compares favorably to CVC4 extended with induction (compare rows ‘Eldarica_noADT’ and ‘Z3_noADT’ to row ‘CVC4+Ind’). An interesting point is that the effectiveness of CVC4 may be increased if one extends the problem formalization with extra axioms, and in particular, with lemmas which may be used for proving the main conjecture. Indeed, CVC4 solves 100 problems when auxiliary lemmas are added, and 134 problems when, in addition, CVC4 runs on the more sophisticated dti encoding, which is based on a double representation of natural numbers, using both the built-in type Int and the ADT definition with the zero and successor constructors. Our results show that in most cases ADTREM needs neither those extra axioms nor that sophisticated encoding.

Finally, in Table 2 we report some problems solved by ADTREM that are not solved by CVC4 with induction, or vice versa.

Table 1. Experimental results. For each problem we have set a timeout limit of 300 seconds. Experiments have been performed on an Intel Xeon CPU E5-2640 2.00GHz with 64GB RAM under CentOS.

|        | CLAM | HipSpec | IsaPlanner | Leon | Total |
|--------|------|---------|------------|------|-------|
| # of problems | 53   | 11      | 63         | 42   | 169   |
| Eldarica | 0    | 2       | 4          | 9    | 15    |
| Z3      | 6    | 0       | 2          | 10   | 18    |
| R       | (18) | 36      | (2) 4      | (56) | 59    | (18) | 30    | (94) | 129   |
| Eldarica_noADT | (18) | 32      | (2) 4      | (56) | 57    | (18) | 29    | (94) | 122   |
| Z3_noADT | (18) | 29      | (2) 3      | (55) | 56    | (18) | 26    | (93) | 114   |
| CVC4+Ind | 17   | 5       | 37         | 15   | 74    |

Table 2. Comparison between ADTREM (with Eldarica) and CVC4+Ind (with any encoding) on some problems. For the details, see [https://fmlab.unich.it/adtrem/](https://fmlab.unich.it/adtrem/).
7 Related Work and Conclusions

Inductive reasoning is supported, with different degrees of human intervention, by many theorem provers, such as ACL2 [26], CLAM [22], Isabelle [33], HipSpec [6], Zeno [40], and PVS [34], which can be used as tools to formalize and prove properties of programs manipulating ADTs. The combination of inductive reasoning and SMT solving techniques has been exploited by many tools for program verification [29, 36, 38, 41, 43, 44].

Leino [29] 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 obligation to be discharged by Z3. 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 [38] 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 and multiple times, whenever new formulas are generated during the proof search, and to use the partially solved conjecture, to generate lemmas that may help in the proofs of the initial conjecture.

The issue of generating suitable lemmas during inductive proofs has been also addressed by Yang et al. [44] 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 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 [41] 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, which uses catamorphisms (i.e., a restricted class of recursive functions), has been implemented into a solver-agnostic tool, called RADA [36].

In the context of CHCs, Unno et al. [43] have proposed a proof system that combines inductive theorem proving with SMT solving. In this approach, the PDR-based engine of Z3 [20] is used to discharge proof obligations generated by the proof system. The approach has been implemented in the RCaml tool to prove relational properties, such as equivalence, of OCaml programs.

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 removing ADTs, thereby deriving a set of CHCs over basic types (i.e., 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, such as those provided by Eldarica [21] and Z3 (in particular, the Spacer engine [25]). Some of the ideas presented in this paper have been preliminarily explored in some case studies in previous work [11, 12], but there neither formal results nor an automated strategy were presented.
A key success factor of the technique presented here is the introduction of difference predicates, which can be viewed as the transformational counterpart of lemma generation. Indeed, as shown in Section 6, 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 [10].

As future work, we plan to apply our transformation-based verification technique to more complex program properties, such as relational properties [9]. Since our approach is not based on reasoning by structural induction, it would also be interesting to consider programs whose recursive definition does not follow the structural pattern of the ADT they manipulate (such as the quicksort and mergesort programs).

References

1. 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, Computer Aided Verification, Proceedings of the 23rd International Conference CAV '11, Snowbird, UT, USA, July 14–20, 2011, Lecture Notes in Computer Science 6806, pages 171–177. Springer, 2011.
2. 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.
3. 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 - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday, Lecture Notes in Computer Science 9300, pages 24–51, Switzerland, 2015. Springer.
4. A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning, volume I, pages 845–911. North Holland, 2001.
5. A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In N. Piterman and S. Smolka, editors, Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS ’13, Lecture Notes in Computer Science 7795, pages 93–107. Springer, 2013.
6. K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In M. P. Bonacina, editor, Automated Deduction – CADE-24, Proceedings of the 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9–14, 2013, Lecture Notes in Artificial Intelligence 7898, pages 392–406. Springer, 2013.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, POPL ’78, pages 84–96. ACM, 1978.
8. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A Tool for Verifying Programs through Transformations. In Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS ’14, Lecture Notes in Computer Science 8413, pages 568–574. Springer, 2014. Available at: http://www.map.uniroma2.it/VeriMAP.
9. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Relational verification through Horn clause transformation. In X. Rival, editor, Proceedings of the 23rd
10. 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. Special Issue on ICLP ’18.

11. 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, *Proceedings Seventh International Workshop on Verification and Program Transformation, VPT@Programming 2019, Genova, Italy, 2nd April 2019*, volume 299 of *EPTCS*, pages 4–18, 2019.

12. 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, *Proceedings of the Sixth Workshop on Horn Clauses for Verification and Synthesis and Third Workshop on Program Equivalence and Relational Reasoning, HCVS/PERR@ETAPS 2019, Prague, Czech Republic, 6–7th April 2019*, volume 296 of *EPTCS*, pages 48–75, 2019.

13. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS ’08*, Lecture Notes in Computer Science 4963, pages 337–340. Springer, 2008.

14. L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In F. Baader, editor, *Automated Deduction – CADE-19, Proceedings of the 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28 – August 2, 2003*, Lecture Notes in Computer Science 2741, pages 279–283. Springer, 2003.

15. H. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

16. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.

17. 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.

18. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming. Special Issue on the 25th Annual GULP Conference*, 13(2):175–199, 2013.

19. S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’12*, pages 405–416, 2012.

20. K. Hoder and N. Bjørner. Generalized property directed reachability. In A. Cimatti and R. Sebastiani, editors, *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing, SAT ’12*, Lecture Notes in Computer Science 7317, pages 157–171. Springer, 2012.

21. H. Hojjat and P. Rümmer. The ELDARICA Horn solver. In N. Bjørner and A. Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 – November 2, 2018*, pages 1–7. IEEE, 2018.
22. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1):79–111, Mar. 1996.
23. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
24. 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.
25. B. Kafle, J. P. Gallagher, and J. F. Morales. RAHFT: A tool for verifying Horn clauses using abstract interpretation and finite tree automata. In *Computer Aided Verification, Proceedings of the 28th International Conference CAV ’16*, Toronto, ON, Canada, July 17–23, 2016, *Proceedings, Part I*, Lecture Notes in Computer Science 9779, pages 261–268. Springer, 2016.
26. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
27. A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-based model checking for recursive programs. In *Computer Aided Verification, Proceedings of the 26th International Conference CAV ’14*, Vienna, Austria, July 18–22, 2014, Lecture Notes in Computer Science 8559, pages 17–34. Springer, 2014.
28. 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, *Computer Aided Verification, Proceedings of the 25th International Conference CAV ’13*, Saint Petersburg, Russia, July 13–19, 2013, Lecture Notes in Computer Science 8044, pages 846–862. Springer, 2013.
29. K. Leino. Automating induction with an SMT solver. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation, Proceedings of the 13th International Conference VMCAI 2012*, Philadelphia, PA, USA, January 22–24, 2012, pages 315–331. Springer, 2012.
30. X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vuillon. The OCaml system, Release 4.06. Documentation and user’s manual, Institut National de Recherche en Informatique et en Automatique, France, 2017.
31. M. Leuschel and M. Butler. ProB: A model checker for B. In *FME 2003: Formal Methods*, pages 855–874. Springer, 2003.
32. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.
33. T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.
34. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction – CADE-11, Proceedings of the 11th International Conference on Automated Deduction*, Saratoga Springs, NY, USA, June 15–18, 1992, pages 748–752. Springer, 1992.
35. A. Pettorossi and M. Proietti. Totally correct logic program transformations via well-founded annotations. *Higher-Order and Symbolic Computation*, 21:193–234, 2008.
36. 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.
37. M. O. Rabin. Decidable theories. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 595–629. North-Holland, 1977.
38. A. Reynolds and V. Kuncak. Induction for SMT solvers. In D. D’Souza, A. Lal, and K. G. Larsen, editors, *Verification, Model Checking, and Abstract Interpretation, Proceedings of the 16th International Conference VMCAI 2015*, Mumbai,
39. H. Seki. On inductive and coinductive proofs via unfold/fold transformations. In D. De Schreye, editor, *Proceedings of the 19th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR ’09)*, Coimbra, Portugal, September 9–11, 2009, Lecture Notes in Computer Science 6037, pages 82–96. Springer, 2010.

40. W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In C. Flanagan and B. König, editors, *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS ’12*, Tallinn, Estonia, March 24 – April 1, 2012, pages 407–421. Springer, 2012.

41. P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In E. Yahav, editor, *Proceedings of the 18th International Symposium on Static Analysis, SAS ’11*, Lecture Notes in Computer Science 6887, pages 298–315. Springer, 2011.

42. H. Tamaki and T. Sato. A generalized correctness proof of the unfold/fold logic program transformation. Technical Report 86-4, Ibaraki University, Japan, 1986.

43. H. Unno, S. Torii, and H. Sakamoto. Automating induction for solving Horn clauses. In R. Majumdar and V. Kuncak, editors, *Computer Aided Verification, Proceedings of the 29th International Conference CAV ’17*, Heidelberg, Germany, Part II, Lecture Notes in Computer Science 10427, pages 571–591. Springer, 2017.

44. W. Yang, G. Fedyukovich, and A. Gupta. Lemma synthesis for automating induction over algebraic data types. In T. Schiex and S. de Givry, editors, *Principles and Practice of Constraint Programming, Proceedings of the 25th International Conference CP 2019*, Stamford, CT, USA, September 30 – October 4, 2019, Lecture Notes in Computer Science 11802, pages 600–617. Springer, 2019.
8 Appendix

In this appendix we show the proofs of the results presented in Sections 4 and 5. First, we recall some definitions and facts from the literature [35].

A reverse-implication-based transformation sequence is a sequence \( P_0 \Rightarrow P_1 \Rightarrow \ldots \Rightarrow P_n \) of sets of clauses where, for \( i = 0, \ldots, n-1 \), \( P_{i+1} \) is derived from \( P_i \) by applying one of the following rules (see Section 4): Definition (Rule R1), Unfolding (Rule R2), Folding (Rule R3), and the following rule, called Body Weakening (Rule W).

(Rule W) Body Weakening. Let \( C : H \leftarrow C_1, C_2, G_L, G_R \) be a clause in \( P_i \), and suppose that the following condition holds for some constant \( c_2 \) and conjunction \( G_2 \) of atoms:

\[
M(\text{Definite}(P_0) \cup \text{Defs}_i) \models (c_1 \land G_1 \rightarrow \exists Y. c_2 \land G_2)
\]

where \( Y = \text{vars}(\{c_2, G_2\}) \setminus \text{vars}(\{H, c, G_L, G_R\}) \). Suppose also that \( \ell(H) > \ell(A) \), for every atom \( A \) occurring in \( G_2 \) and not in \( G_1 \). By body weakening, from clause \( C \) we derive clause \( D : H \leftarrow c_2, G_L, G_R \) and \( P_i \Rightarrow (P_i \setminus \{C\}) \cup \{D\} \).

Theorem 3. If \( P_0 \Rightarrow \ldots \Rightarrow P_n \) is a reverse-implication-based transformation sequence for which Condition (U) of Theorem 1 holds. For \( i = 0, \ldots, n \), let \( D_i = \text{Definite}(P_i) \). Then \( M(P_0 \cup \text{Defs}_i) \subseteq M(D_i) \).

The proof of this theorem [35] is based on the fact that, for \( i = 0, \ldots, n-1 \), \( M(D_0 \cup \text{Defs}_i) \models D_i \leftarrow D_{i+1} \), hence the term reverse-implication-based transformation sequence. Note that, in particular, body weakening replaces a conjunction in the body of a clause \( C \) by a new conjunction which is implied (in \( M(D_0 \cup \text{Defs}_i) \)) by the old one, and thus \( M(D_0 \cup \text{Defs}_i) \models C \leftarrow D \).

Theorem 4. Suppose that \( P_0 \Rightarrow \ldots \Rightarrow P_n \) is a reverse-implication-based transformation sequence and Condition (U) of Theorem 1 holds. 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 \( M \) is a \( \mathbb{D} \)-model of \( P_0 \), then the \( \mathbb{D} \)-interpretation \( M \cup \{\text{newp}(a_1, \ldots, a_k) \mid \text{newp} \) is a head predicate in \( \text{Defs}_n \) and \( a_1, \ldots, a_k \) are ground terms\} is a \( \mathbb{D} \)-model of \( P_0 \cup \text{Defs}_n \), and (ii) if \( M \) is a \( \mathbb{D} \)-model of \( P_0 \cup \text{Defs}_n \), then all clauses of \( P_0 \) and \( \text{Defs}_n \) are true in \( M \), and hence \( M \) is a \( \mathbb{D} \)-model of \( P_0 \).

Then, let us consider a new sequence \( P'_0 \Rightarrow \ldots \Rightarrow P'_n \) obtained from the transformation sequence \( P_0 \Rightarrow \ldots \Rightarrow P_n \) by replacing each occurrence of \( f \) in the head of a clause by a new predicate symbol \( f \). \( P'_0, \ldots, P'_n \) are sets of definite clauses, and thus for \( i = 0, \ldots, n \), \( \text{Definite}(P'_i) = P'_i \). The sequence \( P'_0 \Rightarrow \ldots \Rightarrow P'_n \) satisfies the hypotheses of Theorem 3 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 3, \( f \notin M(P'_0) \cup \text{Defs}_n \) 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, (i) in order to prove Theorem 1 of Section 4 stating the soundness of Rules R1–R7, we show that Rules R4–R7 are all instances of Rule W.
An application of Rule R4 (Clause Deletion) whereby clause $C: H \leftarrow c, G$ is deleted whenever the constraint $c$ is unsatisfiable, is equivalent to the replacement of the body of clause $C$ by false. Indeed, if $c$ is unsatisfiable, we have that:

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

and the applicability condition of Rule W trivially holds. Also the condition:

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

trivially holds, because there are no atoms in false. Thus, the replacement of the body of clause $H \leftarrow c, G$ by false can be performed by applying Rule W.

Let us now consider Rule R5 (Functionality). Recall that by $F(X; Y)$ we denote a conjunction of atoms that defines a total functional relation from $X$ to $Y$. When Rule R5 is applied whereby a conjunction $F(X; Y), F(X; Z)$ is replaced by $Y = Z, F(X; Y)$, it is the case that

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

holds. When this replacement is performed, also the applicability condition of Rule W on the levels of atoms trivially holds, and thus Rule R5 is an instance of Rule W.

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, Rule R6 is an instance of Rule W.

Rule R7 (Differential Replacement), is an instance of Rule W as a consequence of the following lemma.

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

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

where: (i) $W \cap \text{vars}(C) = \emptyset$, and (ii) $\Delta \models \forall (c \rightarrow d)$, 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,

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

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

$$M \models \forall (c \land F(X; Y) \rightarrow \exists W. R(V; W)) \quad (1)$$

Since $W$ does not occur in $C$, from definition $D$, we get

$$M \models \forall (d \land F(X; Y) \land R(V; W) \rightarrow \text{diff}(Z)) \quad (2)$$

From (1) and (2), by (ii), we get the thesis. □

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 $R(V; W), \text{diff}(Z)$, whenever Condition $\ell(H) > \ell(\text{diff})$ holds, is an instance of Rule W.

From the above properties of Rules R4–R7 (including Lemma 1) which relate them to Rule W, we get the following lemma.

**Lemma 2.** Suppose that $P_0 \Rightarrow \ldots \Rightarrow P_n$ is a transformation sequence using Rules R1–R7, and suppose also that Condition (U) of Theorem holds. Then $P_0 \Rightarrow \ldots \Rightarrow P_n$ is a reverse-implication-based transformation sequence.
Now Theorem 1 of Section 4 follows from Lemma 2 and Theorem 4.

Finally, we prove the soundness of Algorithm $R$, that is, Theorem 4 of Section 5. Each procedure used in Algorithm $R$ consists of a sequence of applications of Rules R1–R7. Thus, the thesis follows from Theorem 4 if Condition (U) of the hypothesis of that theorem holds. This condition is satisfied if for each application of the Unfold procedure, the nonempty set of source atoms that are marked as unfoldable in the body of a clause, includes at least one atom with the same level as the head of the clause. This property can always be enforced by dynamically constructing the level mapping $\ell$ during the execution of Algorithm $R$. 