Lemma Generation for Horn Clause Satisfiability: A Preliminary Study

Emanuele De Angelis
DEC, University "G. d'Annunzio" of Chieti-Pescara
Viale Pindaro 42, 65127 Pescara, Italy
emanuele.deangelis@unich.it

Fabio Fioravanti
DEC, University "G. d'Annunzio" of Chieti-Pescara
Viale Pindaro 42, 65127 Pescara, Italy
fabio.fioravanti@unich.it

Alberto Pettorossi
University of Roma Tor Vergata
Via del Politecnico 1, 00133 Roma, Italy
pettorossi@info.uniroma2.it

Maurizio Proietti
CNR-IASI
Via dei Taurini 19, 00185 Roma, Italy
maurizio.proietti@iasi.cnr.it

It is known that the verification of imperative, functional, and logic programs can be reduced to the satisfiability of constrained Horn clauses (CHCs), and this satisfiability check can be performed by using CHC solvers, such as Eldarica and Z3. These solvers perform well when they act on simple constraint theories, such as Linear Integer Arithmetic and the theory of Booleans, but their efficacy is very much reduced when the clauses refer to constraints on inductively defined structures, such as lists or trees. Recently, we have presented a transformation technique for eliminating those inductively defined data structures, and hence avoiding the need for incorporating induction principles into CHC solvers. However, this technique may fail when the transformation requires the use of lemmata whose generation needs ingenuity. In this paper we show, through an example, how during the process of transforming CHCs for eliminating inductively defined structures one can introduce suitable predicates, called difference predicates, whose definitions correspond to the lemmata to be introduced. Through a second example, we show that, whenever difference predicates cannot be introduced, we can introduce, instead, auxiliary queries which also correspond to lemmata, and the proof of these lemmata can be done by showing the satisfiability of those queries.

1 Introduction

In recent years, it has been shown that the verification of program properties can be performed by proving the satisfiability of sets of *constrained Horn clauses* (CHCs). Since a general decision procedure for proving satisfiability of CHCs does not exist, the best one can do is to propose heuristics, and indeed various heuristics for proving satisfiability have been proposed in the literature. Among them we recall: (i) Counterexample Guided Abstraction Refinement (CEGAR) [4], (ii) Craig interpolation [18], and (iii) Property Directed Reachability (PDR) [2, 14]. Moreover, a variety of tools for satisfiability proofs, called *CHC solvers*, has been made available to the scientific community. Let us mention: Eldarica [15], HSF [13], RAHFT [16], VeriMAP [6], and Z3 [20]. Most of those tools work well on simple constraint theories, such as the theory of Linear Integer Arithmetic (LIA) and the theory of Booleans (Bool).

Unfortunately, when the properties to be verified refer to programs that act on inductively defined data structures, such as lists or trees, then the satisfiability proofs via CHC solvers become much harder, or even impossible, because those solvers do not usually incorporate induction principles relative to the data structures in use.

A.P. Lisitsa and A.P. Nemytykh (Eds.): VPT 2019
EPTCS 299, 2019, pp. 4–18, doi:10.4204/EPTCS.299.4

To avoid this difficulty, two approaches have recently been suggested. The first one consists in the incorporation into the CHC solvers of appropriate induction principles [22, 25]. The second one consists in transforming the given set of CHCs into a new set where inductively defined data structures are removed, and whose satisfiability implies the satisfiability of the original clauses [10, 19].

In this paper we will follow this second approach and, in particular, we will consider the *Elimination Algorithm* presented in a previous work of ours [10], which implements a transformation strategy for removing inductively defined data structures and is based on the familiar fold/unfold rules [12, 24]. Thus, if the clauses derived by the Elimination Algorithm have all their constraints in the LIA or Bool theory, then there is no need to modify the CHC solvers for performing the required satisfiability proofs.

Similarly to the case of proof techniques that use induction, the success of the Elimination Algorithm may depend on the discovery of suitable auxiliary lemmata to be used during transformation. The contribution of this paper is a novel technique for generating those lemmata, thereby providing a way of overcoming some difficulties which have been reported in the literature and, in particular, the need of dealing with formulas with second order variables and second order unification, when performing inductive proofs [3].

This novel technique is presented through two examples. In the first example, we show that the introduction of suitable predicates, which we call *difference predicates*, may allow us to perform the desired transformations. A difference predicate expresses the relation between the values computed by two different functions (hence its name), and its definition corresponds to the statement of a lemma which should be proved if one were to show the properties of interest by structural induction. In this example it is demonstrated that, by extending the Elimination Algorithm with the introduction of difference predicates, one can remove inductively defined data structures, thus allowing the completion of the desired proof in many cases where the plain Elimination Algorithm would not terminate.

In the second example, we show how to introduce, during CHC transformation, some implications which correspond to suitable lemmata. Those implications are then proved by showing the satisfiability of auxiliary queries (that is, clauses whose head is `false`), which are derived from the implications.

The paper is structured as follows. In Sections 2 and 3, we present the verification of a property of a functional program that acts on lists of integers, by first (i) deriving by transformation, introducing a suitable difference predicate, a set of CHCs on LIA constraints only (that is, constraints on lists will no longer be present), and then (ii) proving the satisfiability of the derived CHCs by using the solver Z3 acting on LIA constraints only. Note that neither Z3 nor Eldarica are able to check satisfiability of the clauses which are obtained by the direct translation into CHCs of the functional program and the property, before the transformation of Step (i). In Section 4, we present a second example of our verification technique where, during Step (i), we introduce, instead of difference predicates, suitable auxiliary queries. Finally, in Section 5, we comment on the soundness and the mechanization of our verification technique.

2 Horn Clause Satisfiability for Program Verification

Let us consider the following functional program *InsertionSort*, written in the OCaml syntax [17]:

```
type list = Nil | Cons of int * list
let rec ins i l =
  match l with
  | Nil -> Cons(i,Nil)
  | Cons(x,xs) -> if i<=x then Cons(i,Cons(x,xs)) else Cons(x,ins i xs)
```

```
let rec insertionSort l =
  match l with
  | Nil -> Nil
  | Cons(x,xs) -> ins x (insertionSort xs)
let rec sumlist l =
  match l with
  | Nil -> 0
  | Cons(x,xs) -> x + sumlist xs
```

In this program: (i) the `insertionSort` function sorts a list of integers, in ascending order, according to the familiar insertion sort algorithm, and (ii) the `sumlist` function computes, given a list of integers, the sum of all integers in that list.

Let us suppose that for the program *InsertionSort* we want to prove the following Property *Sum* stating that the sum of the elements of a list `l` is equal to the sum of the elements of the sorted list `insertionSort l`. Thus, in formulas, we want to prove that:

$$\forall \mathtt{l}.\ \mathtt{sumlist}\,\mathtt{l} = \mathtt{sumlist}\,(\mathtt{insertionSort}\,\mathtt{l}) \qquad \text{(Property } Sum\text{)}$$

If we want to make a proof of Property *Sum* by induction on the structure of the list `l`, we have to use a lemma stating that the sum of the elements of the list `ins x l` obtained by inserting the element `x` in the list `l` is obtained by adding `x` to the sum of the elements of `l`. This lemma can be expressed by the following formula:

$$\forall \mathtt{x},\mathtt{l}.\ \mathtt{sumlist}\,(\mathtt{ins}\,\mathtt{x}\,\mathtt{l}) = \mathtt{x} + (\mathtt{sumlist}\,\mathtt{l}) \qquad \text{(Lemma } L\text{)}$$

The technique we present in this paper for the proof of Property *Sum*, avoids the explicit introduction of this lemma, and thus the use of the induction principle on lists.

Let us start off by considering the translation of the functional program *InsertionSort* and Property *Sum* into a set of CHCs as explained in the literature [10, 25]. In our example, by that translation we get the following set of clauses [1]:

```
1. false :- M≠N, sumlist(L,M), insertionSort(L,SL), sumlist(SL,N).
2. sumlist([],0).
3. sumlist([X|Xs],M) :- M=X+N, sumlist(Xs,N).
4. ins(I,[],[I]).
5. ins(I,[X|Xs],[I,X|Xs]) :- I≤X.
6. ins(I,[X|Xs],[X|Ys]) :- I>X, ins(I,Xs,Ys).
7. insertionSort([],[]).
8. insertionSort([X|Xs],SL) :- insertionSort(Xs,SXs), ins(X,SXs,SL).
```

In these clauses, $\mathtt{sumlist}(\mathtt{L},\mathtt{M})$, $\mathtt{insertionSort}(\mathtt{L},\mathtt{SL})$, and $\mathtt{ins}(\mathtt{X},\mathtt{L},\mathtt{L1})$ hold iff $\mathtt{sumlist}\,\mathtt{L} = \mathtt{M}$, $\mathtt{insertionSort}\,\mathtt{L} = \mathtt{SL}$, and $\mathtt{ins}\,\mathtt{X}\,\mathtt{L} = \mathtt{L1}$, respectively, hold in program *InsertionSort*.

As usual, we assume that all clauses are universally quantified in front. Clause `1`, also called a *query*,[2] translates Property *Sum* as it stands (using the functional notation) for:

$$\forall \mathtt{l},\mathtt{m},\mathtt{sl},\mathtt{n}.\ \mathtt{sumlist}\,\mathtt{l} = \mathtt{m} \wedge \mathtt{insertionSort}\,\mathtt{l} = \mathtt{sl} \wedge \mathtt{sumlist}\,\mathtt{sl} = \mathtt{n} \ \rightarrow\ \mathtt{m} = \mathtt{n}$$

and clauses `1`–`8` are satisfiable iff Property *Sum* holds. Unfortunately, state-of-the-art CHC solvers, such as Eldarica or Z3, fail to prove satisfiability of clauses `1`–`8`, because those CHC solvers do not incorporate any induction principle on lists.

[1] We use Prolog-like syntax for writing clauses, instead of the more verbose SMT-LIB syntax. The predicates = (equal), ≠ (not-equal), ≤ (less-or-equal), and > (greater) denote constraints between integers.

[2] In the context of Horn clauses, a query (or a goal) is a clause whose head is `false`.

Moreover, starting from clauses `1`–`8`, the Elimination Algorithm [10] which has the objective of eliminating lists from CHCs, is not able to derive a set of clauses without lists, and this inability is due to the fact that the algorithm is unable to introduce a predicate definition corresponding to the needed Lemma *L*.

3 Introducing Difference Predicates

In this section we show that, if we extend the Elimination Algorithm [10] by a technique for introducing the so-called difference predicates, we are able to transform clauses `1`–`8` into a set of clauses without list variables. We will see that the definition of the difference predicate we will introduce exactly corresponds to Lemma *L*.

First, we briefly recall the Elimination Algorithm which makes use of the well-known transformation rules: (i) *define*, (ii) *fold*, (iii) *unfold*, and (iv) *replace* for CHCs [12, 24]. The details can be found in the paper where the algorithm was originally presented [10].

We assume that the *basic types* are the integers and the booleans. We say that a clause has basic types if all its variables have basic types. In the outline of the algorithm below, *Cls* is the set of input clauses which define the predicates occurring in the given set *Qs* of input queries. *Defs* is the set of definition clauses which are introduced by the algorithm and used for folding. *Defs* accumulates the sets *NewDefs* of definition clauses which are introduced during the various iterations of the while-do loop. *FldCls*, *UnfCls*, and *RCls* are the sets of clauses which are obtained after the applications of the folding, unfolding, and replace rules, respectively. The Elimination Algorithm works by enforcing that all new predicate definitions which are introduced have arguments of basic types only (and thus, no list variables will occur in their arguments).

The Elimination Algorithm $\mathscr{E}$.
Input: A set $Cls \cup Qs$, where *Cls* is a set of constrained Horn clauses and *Qs* is a set of queries.
Output: A set *TransfCls* of clauses such that: (i) $Cls \cup Qs$ is satisfiable iff *TransfCls* is satisfiable, and (ii) every clause in *TransfCls* has basic types.

$Defs := \emptyset$; $InCls := Qs$; $TransfCls := \emptyset$;
while $InCls \neq \emptyset$ **do**
  Define-Fold$(Defs, InCls, NewDefs, FldCls)$;
  Unfold$(NewDefs, Cls, UnfCls)$;
  Replace$(UnfCls, Cls, RCls)$;
  $Defs := Defs \cup NewDefs$; $InCls := RCls$; $TransfCls := TransfCls \cup FldCls$;

In order to get clauses without list variables, we start off the transformation of clauses `1`–`8` by applying the Elimination Algorithm $\mathscr{E}$ to the following sets of clauses and queries: $Cls = \{$clause `2`, ..., clause `8`$\}$ and $Qs = \{$query `1`$\}$. Thus, the first step is the introduction of a new predicate `new1` by the following clause (here and in what follows the numbers under the atoms of the body of the clauses identify the individual atoms):

```
9. new1(M,N) :- sumlist(L,M), insertionSort(L,SL), sumlist(SL,N).
                    (9.1)             (9.2)            (9.3)
```

The arguments of `new1` are the integer variables occurring in the body of clause `9`. Thus, by folding query `1`, we derive a new clause without occurrences of list variables:

```
10. false :- M≠N, new1(M,N).
```

We proceed by eliminating lists from the body of clause 9. By unfolding clause 9, we replace the predicate calls by their definitions and we derive the following two clauses:

```
11. new1(0,0).
12. new1(M1,N1) :- M1=H+M, sumlist(T,M), insertionSort(T,ST), ins(H,ST,SU),
                            (12.1)          (12.2)             (12.3)
                          sumlist(SU,N1).
                            (12.4)
```

Now, in order to fold clause 12 using clause 9 and derive a recursive definition of new1, we depart from the Elimination Algorithm and we propose a new technique that introduces a so-called *difference predicate* diff. The definition of diff is based on the mismatch between clause 9 and clause 12. The new technique is applied according to the following six steps.

• *Step* 1. *Embed*. We have that the body of clause 9 is *embedded* in the body of clause 12, that is, each distinct atom in the body of clause 9 is a variant of a distinct atom in the body of clause 12. In particular, (i) 9.1 is a variant of 12.1, (ii) 9.2 is a variant of 12.2, and (iii) 9.3 is a variant of 12.4. However, clause 12 cannot be folded using clause 9, because the conjunction (9.1, 9.2, 9.3) does not match the conjunction (12.1, 12.2, 12.4) (see arguments SL, ST, and SU). It can be shown that no further unfolding of clause 12 will generate a clause whose body is an instance of the body of clause 9, and indeed the Elimination Algorithm will not terminate.

• *Step* 2. *Rename*. We rename apart clause 9, which we would like to use for folding, so as to have variable names that do not occur anywhere else[3]. We get:

```
9a. new1(Ma,Na) :- sumlist(La,Ma), insertionSort(La,SLa), sumlist(SLa,Na).
                      (9a.1)                (9a.2)               (9a.3)
```

• *Step* 3. *Match*. We match the body of clause 9a against the body of clause 12 to be folded. We match the conjunction (9a.1, 9a.2) with the conjunction (12.1, 12.2) by the renaming substitution $\sigma = \{$La/T, Ma/M, SLa/ST$\}$, but we cannot extend this matching to the remaining atom 9a.3 because the substitution $\{$SLa/SU, Na/N1$\}$ is inconsistent with σ. By applying the substitution σ, we get the following clause 9m, which is a variant of clause 9a:

```
9m. new1(M,Na) :- sumlist(T,M), insertionSort(T,ST), ‖ sumlist(ST,Na).
                    (9m.1)          (9m.2)                (9m.3)
```

This clause 9m is the actual clause which we will use for folding at Step 6 below. The marker ‖ we have placed in its body has no logical meaning and it is used only as a separator between the *matching conjunction* (9m.1, 9m.2) to its left and the *mismatching conjunction* 9m.3 to its right (in general, also the mismatching conjunction may consist of more than one atom).

Also for the clause to be folded (clause 12 in our case) we define the matching and the mismatching conjunctions: (i) the *matching conjunction* of the clause to be folded is equal to the one of the clause we will use for folding (atoms 12.1 and 12.2 in our case), while (ii) the *mismatching conjunction* of the clause to be folded is the conjunction of its body atoms (atoms 12.3 and 12.4 in our case) that do not belong to the matching conjunction.

• *Step* 4. *Introduce a Difference Predicate*. Now, in order to fold clause 12 using clause 9m, we need to replace some atoms of the body of clause 12. In particular,

[3] This is the only variable renaming that we perform during Steps 1–6.

Point (4.1): we have to remove from the body of clause `12` its mismatching conjunction (atoms `12.3` and `12.4`), and
Point (4.2): we have to add to the body of clause `12` the mismatching conjunction of clause `9m` (atom `9m.3`).

Indeed, if we perform the actions of Points (4.1) and (4.2) (see Step 5 below), then we can fold clause `12` by using clause `9m` (see Step 6 below). However, before performing that folding step, we also add to the body of clause `12` a new atom defining a so-called *difference predicate*. This new atom expresses the relation between the output non-list variables [4] of the conjunction to be removed according to Point (4.1) (that is, the integer variable `N1`, in our case), and the output non-list variables of the conjunction to be added according to Point (4.2) (that is, the integer variable `Na`, in our case). The difference predicate we introduce, called `diff`, is defined by a clause, whose body is made out of: (i) the mismatching conjunction to be removed, and (ii) the mismatching conjunction to be added. Thus, we get the following definition for `diff`:

```
13. diff(H,Na,N1) :- ins(H,ST,SU), sumlist(SU,N1), sumlist(ST,Na).
                      (12.3)        (12.4)           (9m.3)
```

The arguments `H`, `Na`, and `N1` of the head are the non-list variables occurring in the body (obviously these arguments can be placed in any order) [5]. Clause `13` can be read (in functional notation) as follows:

if `sumlist (ins H ST)` = `N1` and `sumlist ST` = `Na`,
then `diff(H,Na,N1)` expresses the relation between the output integer variables `Na` and `N1` (as a function of the input integer variable `H`).

• *Step* 5. *Replace*. In the clause to be folded (clause `12` in our case) we replace the mismatching conjunctions (atoms `12.3` and `12.4` in our case) by: (i) the mismatching conjunction of the clause we will use for folding (atom `9m.3` in our case), and (ii) the head of the clause defining the difference predicate (clause `13` in our case). By doing so, from clause `12` we get the following clause:

```
12r. new1(M1,N1) :- M1=H+M, sumlist(T,M), insertionSort(T,ST), sumlist(ST,Na),
                    diff(H,Na,N1).
```

• *Step* 6. *Fold*. We fold clause `12r` using clause `9m` (this folding is possible due to Step 5) and we get:

```
12f. new1(M1,N1) :- M1=H+M, new1(M,Na), diff(H,Na,N1).
```

Now let us briefly discuss the correctness of the above transformation consisting of Steps 1–6.

In general, we say that a transformation from a given set S_1 of clauses to a derived set S_2 of clauses is *sound* whenever:

if S_2 is satisfiable, *then* S_1 is satisfiable. (*Soundness*)

Thus, the satisfiability of the derived set of clauses is sufficient to guarantee the property of the functional program that we want to verify (Property *Sum*, in our example).

In the case of our transformation Steps 1–6, soundness is ensured by the fact that, at Step 5, we replace a given conjunction of atoms by a new conjunction which is implied by the given one. For instance, in our *InsertionSort* example, we have that:

I. $\forall$(`ins(H,ST,SU)`, `sumlist(SU,N1)` $\rightarrow$ ($\exists$`Na.` `sumlist(ST,Na)`, `diff(H,Na,N1)`))

[4] The output variables of the predicates `ins` and `sumlist` in the atoms `12.3`, `12.4`, and `9m.3` are defined as expected, when considering the associated functional expressions `(ins H ST) = SU`, `(sumlist SU) = N1`, and `(sumlist ST) = Na`, respectively.
[5] Note that this choice of the arguments of `diff` is in accordance with our goal of eliminating list variables.

where, for any formula φ, $\forall(\varphi)$ denotes the *universal closure* of φ, that is, the formula $\forall\overline{\mathtt{X}}.\,\varphi$, where $\overline{\mathtt{X}}$ is the tuple of the variables occurring free in φ. Indeed, formula I holds in the least model of clauses 2 – 8, because: (i) for all L and N, sumlist(L,N) defines a total functional relation from (the domain of) L to (the domain of) N (and thus, $\exists$Na. sumlist(ST,Na) is true), and (ii) the predicate diff is defined by clause 13.

It can also be shown that the above Steps 1– 6 preserve satisfiability of clauses in the following stronger sense:

if (H1) ins(X,S,S1) and sumlist(L,N) define total functional relations, (*Equisatisfiability*)
and
(H2) diff(H,Na,N1) is a functional relation from the pair (H,Na) of integers to the integer N1,
then the replacement of clause 12 by clauses 12f and 13 produces an *equisatisfiable* set of clauses.

Indeed, under Hypotheses (H1) and (H2), also the converse of implication I holds.

Note that Hypothesis (H1) holds by construction, because the predicates ins and sumlist come from the translation of functional programs that terminate for all input values.

The detailed proofs of general results concerning the soundness and equisatisfiability properties of our transformations are outside the scope of the present paper.

The clauses we have derived so far are clauses 10, 11, 12f, 13, together with the clauses defining the predicates sumlist and ins, that is, clauses 2 – 6. Still clause 13, which defines the predicate diff, and clauses 2 – 6 have list variables and we should eliminate them by applying the Elimination Algorithm. If we do so by starting from *Cls* = {clause 2, ..., clause 6} and *InCls* = {clause 13}, we derive the following final clauses (during this elimination there is no need of introducing any new difference predicate):

```
10.  false :- M≠N, new1(M,N).
11.  new1(0,0).
12f. new1(M1,N1) :- M1=H+M, new1(M,Na), diff(H,Na,N1).
14.  diff(H,0,N1) :- N1=H.
15.  diff(H,Na,N1) :- H≤X, Na=X+N2, N1=H+Na, new2(N2).
16.  diff(H,Na,N1) :- H>X, Na=X+N2, N1=X+N3, diff(H,N2,N3).
17.  new2(0).
18.  new2(N) :- N=X+N1, new2(N1).
```

This final set of clauses has no list arguments and all the constraints belong to the LIA theory. The CHC solver Z3 [6] proves that this set of clauses is satisfiable by computing the following model which is expressible in LIA:

D1. new1(M,N) ≡ M=N
D2. new2(N) ≡ true
D3. diff(H,Na,N1) ≡ H+Na=N1

Indeed, by replacing the left-hand side atoms by the corresponding right-hand side LIA formulas in the final set of clauses 10, 11, 12f and 14 –18, we get a set of valid implications. Note that, by D3 we have that diff(H,Na,N1) is a functional relation (that is, the usual '+' on integers) from (H,Na) to N1.

Thus, we have proved that Property *Sum* holds for the given program *InsertionSort*.

[6] By first translating CHCs from Prolog syntax to SMT-LIB syntax and then using the command: 'z3_4.8.4 -smt2 sumlist.transf.smt fp.engine=spacer dump_models=true'.

Let us conclude this section by commenting on the relationship between difference predicates and lemmata. If in clause 13 defining the difference predicate `diff` we replace its head `diff(H,Na,N1)` by the constraint `H + Na = N1` computed by the solver Z3, we exactly get the CHC translation of Lemma *L* needed for proving Property *Sum* by structural induction on lists. Thus, the introduction of difference predicates can be viewed, at least in some cases, as a way of generating the lemmata required during proofs by structural induction.

4 Introducing Auxiliary Queries

In this section, we show through an example that during the transformation Step 4 presented in Section 3, we can introduce, instead of difference predicates, some auxiliary queries which correspond to lemmata required in the proof of the property of interest.

Let us consider the following functional program *Rotate*, written in the OCaml syntax, which defines: (i) the familiar `append` function which concatenates two lists, (ii) the `len` function which computes the length of a list, and (iii) the `rotate` function which computes the circular rotation of a given list by `m` ($\geq$ `0`) positions. For instance, `rotate 2 [7,4,5,9,1] = [5,9,1,7,4]`.

```
type list = Nil | Cons of int * list
let rec append l1 l2 =
   match l1 with
   | Nil -> l2
   | Cons(h,t) -> Cons(h, append t l2)
let rec len l =
   match l with
   | Nil -> 0
   | Cons(h,t) -> 1 + len t
let rec rotate m l =
  if m<=0 then l else
    match l with
    | Nil -> Nil
    | Cons(h,t) -> rotate (m-1) (append t (Cons(h,Nil)))
```

Let us suppose that we want to prove the following Property *Rotation* stating that:

$\forall$`l,k.` `rotate (len l) (append l k) = append k l` (Property *Rotation*)

This property is used as a running example in a paper by Alan Bundy [3] on the automation of proofs by mathematical induction. In that paper the author discusses the issue of how to generate the lemmata needed for the inductive proofs and suggests the introduction of formulas with second order variables. Unfortunately, those second order variables require the use of second order unification and narrowing (for these concepts the reader may refer to the paper by Baader and Snyder [1]).

Now we will see how, with the help of the `rotate` example, the difficulties due to the use of second order variables can be overcome and the required lemmata can be generated by applying a variant of the technique proposed in Section 3. In particular, Property *Rotation* is translated into a query (see clause 1 below) and the lemmata needed for the satisfiability proof of that query are introduced, not in the form of difference predicates, but in the form of auxiliary queries (see Step 4* below) whose satisfiability should in turn be proved.

As in the example considered in Sections 2 and 3, in the `rotate` example here we start off by translating the initial functional program into a set of CHCs. By doing so we get:

```
1. false :- len(L,M), append(L,K,W), rotate(M,W,Z), append(K,L,Z1), Z≠listZ1.
2. append([],Ys,Ys).
3. append([H|Xs],Ys,[H|Zs]) :- append(Xs,Ys,Zs).
4. len([],0).
5. len([H|T],M) :- M=N+1, len(T,N).
6. rotate(M,L,L) :- M≤0.
7. rotate(M,[],[]) :- M>0.
8. rotate(M,[H|T],Z) :- M>0, N=M-1, append(T,[H],R), rotate(N,R,Z).
```

where query `1` translates Property *Rotation*. For the clauses `2`–`8` we have that `append(Xs,Ys,Zs)`, `len(L,M)`, and `rotate(M,L,L1)` hold iff `append Xs Ys` = `Zs`, `len L` = `M`, and `rotate M L` = `L1`, respectively, hold in the given functional program.

In what follows, we will use the predicates $=_{\texttt{list}}$ and $\neq_{\texttt{list}}$ to denote list equality and disequality, respectively. Now, similarly to Section 3, we would like to transform clauses `1`–`8` into an equisatisfiable set of clauses where no list variables occur. In this transformation we apply the Elimination Algorithm and we start off by introducing a new predicate `new1` using the following clause:

```
9. new1(M) :- len(L,M), append(L,K,W), rotate(M,W,Z), append(K,L,Z1), Z ≠list Z1.
```

where the argument `M` of `new1` is the only integer variable in the body of query `1`. Then, we fold query `1` using clause `9`, and we get:

```
10. false :- new1(M).
```

We proceed by performing some unfolding steps starting from the `len` predicate in clause `9`. After suitable variable renamings we derive the following two clauses:

```
11. new1(0) :- append(A,[],B), A ≠list B.
12. new1(A) :- A=B+1, B≥0, len(C,B), append(C,D,E), append(E,[F],G),
               rotate(B,G,H), append(D,[F|C],I), H ≠list I.
```

The list variables in clause `11` are eliminated by introducing a new predicate `new2`, defined by the clause:

```
13. new2 :- append(A,[],B), A ≠list B.
```

which is then used for folding clause `11`. By folding, we get:

```
11f. new1(0) :- new2.
```

The recursive definition of `new2`, derived by an iteration of the Elimination Algorithm, consists of the following clause only:

```
14. new2 :- new2.
```

Now, in order to fold clause `12` using clause `9` and derive a recursive definition of `new1`, we perform the following six steps, which are similar to those presented in Section 3, with the exception that, as already mentioned, we introduce auxiliary queries, instead of difference predicates.

• *Step* 1. *Embed*. We have that the body of clause `9` is embedded in the body of clause `12`, that is, for each occurrence *A* of an atom in the body of clause `9` there is an occurrence of an atom in the body of clause `12` which is an instance of *A*.

• *Step* 2. *Rename*. We rename apart clause `9`, which we would like to use for folding. We get:

```
9a. new1(Ma) :- len(La,Ma), append(La,Ka,Wa), rotate(Ma,Wa,Za),
                append(Ka,La,Z1a), Za ≠list Z1a.
```

• *Step* 3. *Match.* We match the body of clause `9a` against the body of clause `12` to be folded. We have that the conjunction `len(La,Ma), rotate(Ma,Wa,Za)` in the body of clause `9a` matches the conjunction `len(C,B), rotate(B,G,H)` in the body of clause `12` via the substitution $\sigma = \{$`La/C`, `Ma/B`, `Wa/G`, `Za/H`$\}$. By applying the substitution σ to clause `9a`, we get:

```
9m. new1(B) :- len(C,B), append(C,Ka,G), rotate(B,G,H),
               append(Ka,C,Z1a), H ≠list Z1a.
```

Thus, the mismatching conjunction of clause `9m` is:

```
M. append(C,Ka,G), append(Ka,C,Z1a), H ≠list Z1a.
```

and the mismatching conjunction of clause `12` is:

```
N. append(C,D,E), append(E,[F],G), append(D,[F|C],I), H ≠list I.
```

• *Step* 4*. *Introduce Auxiliary Queries.* Now, in order to fold clause `12` using clause `9m`, we need to replace the mismatching conjunction `N` of clause `12` by the mismatching conjunction `M` of clause `9m`. This replacement cannot be done by applying the technique presented in Section 3, where we have also introduced the difference predicate `diff`. Indeed, no output integer variables occur in the mismatching conjunctions and the associated `diff` predicate would have no arguments at all.

Thus, we will follow an alternative path: (i) first, we will do the replacement of `N` by `M` (see Step 5 below), and then (ii) we will prove, as an auxiliary lemma, the soundness of that replacement. As indicated in Section 3 (see Property `I`), this auxiliary lemma, call it `L1`, is $\forall(\mathtt{N} \rightarrow \exists \overline{\mathtt{Y}}.\mathtt{M})$, where $\overline{\mathtt{Y}}$ is the tuple of the variables occurring in `M` and not in the rest of clause `9m`. By using the definitions of the conjunctions `N` and `M`, Lemma `L1` can be written as follows:

```
L1.  ∀ (append(C,D,E), append(E,[F],G), append(D,[F|C],I), H ≠list I →
           ∃ Ka,Z1a. append(C,Ka,G), append(Ka,C,Z1a), H ≠list Z1a)
```

This universally quantified implication `L1` is not in CHC form, and in order to prove it by using our transformation technique, we first need to transform it into a set of CHCs as indicated in the following Steps (4*.1) – (4*.4).

Step 4*.1. *Move conclusion to premise.* We move the conclusion of `L1` to the premise and we get a new universally quantified implication:

```
H1. ∀ (append(C,D,E), append(E,[F],G), append(D,[F|C],I), H ≠list I,
        (¬∃ Ka,Z1a. append(C,Ka,G), append(Ka,C,Z1a), H ≠list Z1a) → false)
```

Step 4*.2. *Use Functionality and Totality of* `append`*, and Properties of* $=_{\mathtt{list}}$. Since `append(Xs,Ys,Zs)` denotes a total functional relation from `(Xs,Ys)` to `Zs`, from `H1` we get that the following universally quantified equivalence holds:

```
  ∀ ((¬∃ Ka,Z1a. append(C,Ka,G), append(Ka,C,Z1a), H ≠list Z1a)  ↔
     (¬∃ Ka. append(C,Ka,G) ∨
     ( ∃ Ka,Z1a. append(C,Ka,G), append(Ka,C,Z1a), H =list Z1a))
```

Thus, by using the distributive law, we rewrite `H1` into the conjunction of the following two universally quantified implications:

```
H2. ∀ (append(C,D,E), append(E,[F],G), append(D,[F|C],I), H ≠list I,
      (¬∃ Ka. append(C,Ka,G)) → false)
H3. ∀ (append(C,D,E), append(E,[F],G), append(D,[F|C],I), H ≠list I,
      (∃ Ka,Z1a. append(C,Ka,G), append(Ka,C,Z1a), H =list Z1a) → false)
```

Now, since `H` has a single occurrence in the premise of `H2` and $\forall$`I.` $\exists$`H. H` $\neq_{\texttt{list}}$ `I`, we can remove `H` $\neq_{\texttt{list}}$ `I` from `H2`. Then, `I` has a single occurrence in the implication derived after removal and, by the totality of `append`, we can remove also `append(D,[F|C],I)`. Thus, from `H2` we get:

```
H4. ∀ (append(C,D,E), append(E,[F],G), (¬∃Ka. append(C,Ka,G)) → false)
```

Step 4*.3. *Derive CHC Queries.* (i) First we replace ¬∃ `Ka. append(C,Ka,G)` in `H4` by a new predicate `not_exists_2nd_append(C,G)`, whose defining clauses will be derived at the following Step 4*.4, and then (ii) we remove the existential quantification from the premise of `H3` (this removal preserves equivalence). By doing so, we get the following two CHC queries:

```
Q1.1  false :- append(C,D,E), append(E,[F],G), not_exists_2nd_append(C,G).
Q1.2  false :- append(C,D,E), append(E,[F],G), append(D,[F|C],I), H ≠list I,
               append(C,Ka,G), append(Ka,C,Z1a), H =list Z1a.
```

Step 4*.4. *Eliminate Negation.* We derive CHCs defining predicate `not_exists_2nd_append(Xs,Ys)` by using well-known techniques for eliminating negation from logic programs (see, for instance, the *Negation Technique* [23] and the *UFS* transformation strategy [21]):

```
15. not_exists_2nd_append([X|Xs],[]).
16. not_exists_2nd_append([X|Xs],[Y|Ys]) :- X≠Y.
17. not_exists_2nd_append([X|Xs],[Y|Ys]) :- X=Y, not_exists_2nd_append(Xs,Ys).
```

• *Step* 5. *Replace.* Now by applying Lemma `L1`, we replace, in clause `12`, conjunction `N` by conjunction `M`, and we get the following clause:

```
12r. new1(A) :- A=B+1, B≥0, len(C,B), append(C,Ka,G), rotate(B,G,H),
                append(Ka,C,Z1a), H ≠list Z1a.
```

• *Step* 6. *Fold.* We fold clause `12r` using clause `9m`, and we get:

```
12f. new1(A) :- A=B+1, B≥0, new1(B).
```

The clauses derived after Steps 1–6 are: `10`, `11f`, `14`, `12f`, `15`, `16`, `17`, together with queries `Q1.1` and `Q1.2`, and clauses `2`–`8` belonging to the initial set.

Now, we are left with the task of proving Lemma `L1`. Since Steps (4*.1)–(4*.4) preserve satisfiability, this proof can be done by showing the satisfiability of the two queries `Q1.1` and `Q1.2`. Our transformation continues starting from those two queries by following a strategy similar to the one we have applied above starting from the initial query `1`. Thus, we introduce the following two new definitions:

```
18. new3(F) :- append(C,D,E), append(E,[F],G), not_exists_2nd_append(C,G).
19. new4(F) :- append(C,D,E), append(E,[F],G), append(D,[F|C],I), H ≠list I,
               append(C,Ka,G), append(Ka,C,Z1a), H =list Z1a.
```

Then, we fold `Q1.1` and `Q1.2` using those clauses, and we get:

```
20. false :- new3(F).
21. false :- new4(F).
```

After some more transformation steps, we derive the following final set of clauses, called *TransfCls*:

```
10.  false :- new1(A).
11f. new1(0) :- new2.
12f. new1(A) :- A=B+1, B≥0, new1(B).
14.  new2 :- new2.
20.  false :- new3(A).
21.  false :- new4(A).
22.  new3(A) :- new3(A).
23.  new4(A) :- new5(A).
24.  new5(A) :- new5(A).
25.  false :- new6(A,F).
26.  new6(A,B) :- new7(B)
27.  new6(A,B) :- B≠C, new8(C).
28.  new6(A,B) :- new8(B).
29.  new6(A,B) :- new6(A,B).
30.  new7(A) :- B≠A, new8(B).
31.  new7(A) :- new8(A).
32.  new8(A) :- new8(B).
33.  false :- new9(A).
34.  new9(A) :- new9(A).
```

During the derivation of the above clauses we generate the following two extra Lemmata L2 and L3 and we introduce the corresponding two auxiliary queries Q2 and Q3, respectively:

L2. ∀ (append(C,[A,F|B],G), append(I,[F|B],H), G $\neq_{\texttt{list}}$ H →
(∃ F1,G1. append(C,[A|B],F1), append(I,B,G1), F1 $\neq_{\texttt{list}}$ G1))

Q2. false :- append(C,[A,F|B],G), append(I,[F|B],H), G $\neq_{\texttt{list}}$ H,
append(C,[A|B],F1), append(I,B,G1), F1 $=_{\texttt{list}}$ G1.

L3. ∀ (append(B,[A|C],D) → ∃ B1.append(B1,C,D))

Q3. false :- append(B,[A|C],D), not_exists_1st_append(C,D).

where not_exists_1st_append(C,D) is a predicate equivalent to ¬∃ B1.append(B1,C,D).

The final set *TransfCls* of clauses has no list arguments and all constraints are LIA formulas. As in the example of Section 3, it can be shown that the transformation steps we have performed, are sound, and thus if we are able to prove the satisfiability of the clauses of *TransfCls*, then Property *Rotation* holds.

Now, the CHC solver Z3 easily proves that the set *TransfCls* of clauses is satisfiable. Indeed, every clause in *TransfCls* has one atom in its body, and hence the interpretation that assigns the truth value false to every predicate is a model. In particular, the satisfiability of *TransfCls* shows also the validity of the various lemmata we have generated during the derivation.

This concludes the proof that Property *Rotation* holds for the given rotate function.

5 Concluding Remarks

Let us briefly discuss on the soundness of the transformation technique presented in this paper through a couple of examples in Sections 3 and 4 and also on the mechanization of that technique.

The notion of soundness we have used is defined by the following property: if the clauses obtained after transformation are satisfiable, then so are the clauses before transformation. Thus, the satisfiability of the clauses obtained after transformation is sufficient to guarantee that the property of the functional program that we want to verify indeed holds.

The crucial hypothesis needed to show the soundness of our transformation technique is that the predicates occurring in the initial set of clauses define total functional relations. This property is enforced by construction, whenever those predicates are the CHC translation of functions that terminate for all inputs. Moreover, to show soundness, we also use the fact that every lemma generated during derivation is an implication, and we replace, in the body of a clause, an instance of the premise of the lemma by an instance of its conclusion. If the lemmata are not equivalences, the transformations do not necessarily derive final clauses that are equisatisfiable with respect to the initial ones. However, as mentioned at the end of Section 3, equisatisfiability is guaranteed if every generated lemma corresponds to the definition of a *functional* difference predicate. This functionality property can be checked in the model computed by the CHC solver, which is expressed as a set of LIA and Bool constraints.

Concerning the mechanization of our transformation technique, we need to extend the Elimination Algorithm [10] with a suitable automated mechanism for introducing difference predicates and/or auxiliary queries. As shown in Sections 3 and 4, this mechanism can be based on the result of matching the clauses obtained by unfolding (see clause 12 in the *InsertionSort* example, and clause 12 in the *Rotate* example) against the predicate definitions introduced in previous transformation steps (see clause 9 in both examples). More sophisticated mechanisms may take into account the constraints occurring in the clauses, and may apply widening techniques which have been considered in other transformation methods [5, 16]. We have made initial steps towards an implementation of such an extended Elimination Algorithm using the VeriMAP transformation and verification system [6].

In order to evaluate the generality of our verification approach based on the Elimination Algorithm extended with lemma generation, we have also done some experiments on various sorting algorithms and we have semi-automatically proved various properties for a few of them [11].

To summarize, this paper presents ongoing work which follows a very general approach to program verification based on constrained Horn clauses. As shown in the examples we have presented, the reduction of a program verification problem to a CHC satisfiability problem can often be obtained by a straightforward translation. However, proving the satisfiability of the clauses obtained by that translation is, in many cases, a much harder task. In a series of papers [5, 7, 8, 9, 10, 16, 19] it has been shown that by combining various transformation techniques, such as *Specialization* and *Predicate Pairing*, we can derive equisatisfiable sets of clauses where the efficacy of the CHC solvers is significantly improved. This approach avoids the burden of implementing very sophisticated solving strategies depending on the class of satisfiability problems to be solved. In particular, in the class of problems considered in this paper consisting in checking the satisfiability of clauses over inductively defined data structures, we can avoid to implement *ad hoc* strategies that deal with induction proofs. We leave it for future work to experiment on various benchmarks available from the literature and to test whether the transformation-based approach pays off in practice.

6 Acknowledgments

All authors are members of the INdAM-GNCS Italian Research Group. Many thanks to the anonymous referees for their helpful suggestions and constructive comments.

References

[1] F. Baader & W. Snyder (2001): *Unification Theory*. In A. Robinson & A. Voronkov, editors: *Handbook of Automated Reasoning*, I, Elsevier Science, pp. 445–532, doi:10.1016/B978-044450813-3/50010-2.

[2] A. R. Bradley (2011): *SAT-Based Model Checking without Unrolling*. In Ranjit Jhala & David A. Schmidt, editors: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23–25, 2011. Proceedings*, Lecture Notes in Computer Science 6538, Springer, pp. 70–87, doi:10.1007/978-3-642-18275-4_7.

[3] A. Bundy (2001): *The Automation of Proof by Mathematical Induction*. In A. Robinson & A. Voronkov, editors: *Handbook of Automated Reasoning*, I, North Holland, pp. 845–911, doi:10.1016/B978-044450813-3/50015-1.

[4] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu & H. Veith (2000): *Counterexample-Guided Abstraction Refinement*. In E.A. Emerson & A.P. Sistla, editors: *Proceedings of the 12th International Conference on Computer Aided Verification, CAV '00*, Lecture Notes in Computer Science 1855, Springer, pp. 154–169, doi:10.1007/10722167_15.

[5] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2014): *Program Verification via Iterated Specialization*. *Science of Computer Programming* 95, Part 2, pp. 149–175, doi:10.1016/j.scico.2014.05.017. Selected and extended papers from Partial Evaluation and Program Manipulation 2013.

[6] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2014): *VeriMAP: A Tool for Verifying Programs through Transformations*. In E. Ábrahám & K. Havelund, editors: *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, Springer, pp. 568–574, doi:10.1007/978-3-642-54862-8_47. Available at: http://www.map.uniroma2.it/VeriMAP.

[7] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2015): *Proving correctness of imperative programs by linearizing constrained Horn clauses*. *Theory and Practice of Logic Programming* 15(4–5), pp. 635–650, doi:10.1017/S1471068415000289.

[8] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2017): *Semantics-based generation of verification conditions via program specialization*. *Science of Computer Programming* 147, pp. 78–108, doi:10.1016/j.scico.2016.11.002. Selected and Extended papers from the International Symposium on Principles and Practice of Declarative Programming 2015.

[9] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2018): *Predicate Pairing for program verification*. *TPLP* 18(2), pp. 126–166, doi:10.1017/S1471068417000497.

[10] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2018): *Solving Horn Clauses on Inductive Data Types Without Induction*. *TPLP* 18(3-4), pp. 452–469, doi:10.1017/S1471068418000157.

[11] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2019): *Proving Properties of Sorting Programs: A Case Study in Horn Clause Verification*. In E. De Angelis, G. Fedyukovich, N. Tzevelekos & M. Ulbrich, editors: *Proceedings of HCVS/PERR 2019: 6th Workshop on Horn Clauses for Verification and Synthesis and 3rd Workshop on Program Equivalence and Relational Reasoning, Prague, Czech Republic, April 6–7, 2019*, To appear in EPTCS. Available at https://conf.researchr.org/details/etaps-2019/hcvs-2019-papers/3/Proving-Properties-of-Sorting-Programs-A-Case-Study-in-Horn-Clause-Verification.

[12] S. Etalle & M. Gabbrielli (1996): *Transformations of CLP Modules*. *Theoretical Computer Science* 166, pp. 101–146, doi:10.1016/0304-3975(95)00148-4.

[13] S. Grebenshchikov, N. P. Lopes, C. Popeea & A. Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, pp. 405–416, doi:10.1145/2345156.2254112.

[14] K. Hoder & N. Bjørner (2012): *Generalized Property Directed Reachability*. In Alessandro Cimatti & Roberto Sebastiani, editors: *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing, SAT '12*, Lecture Notes in Computer Science 7317, Springer, Berlin, Heidelberg, pp. 157–171, doi:10.1007/978-3-642-31612-8_13.

[15] H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak & P. Rümmer (2012): *A Verification Toolkit for Numerical Transition Systems*. In D. Giannakopoulou & D. Méry, editors: *FM '12: Formal Methods, 18th International Symposium, Paris, France, August 27–31, 2012. Proceedings*, Lecture Notes in Computer Science 7436, Springer, pp. 247–251, doi:10.1007/978-3-642-32759-9_21.

[16] B. Kafle, J. P. Gallagher & J. F. Morales (2016): *RAHFT: A Tool for Verifying Horn Clauses Using Abstract Interpretation and Finite Tree Automata*. In S. Chaudhuri & A. Farzan, editors: *Computer Aided Verification, 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, Lecture Notes in Computer Science 9779, Springer, pp. 261–268, doi:10.1007/978-3-319-41528-4_14.

[17] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy & J. Vouillon (2017): *The OCaml system, Release 4.06*. Documentation and user's manual, Institut National de Recherche en Informatique et en Automatique, France.

[18] K. L. McMillan (2003): *Interpolation and SAT-Based Model Checking*. In Warren A. Hunt Jr. & Fabio Somenzi, editors: *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, Lecture Notes in Computer Science 2725, Springer, pp. 1–13, doi:`10.1007/978-3-540-45069-6_1`.

[19] D. Mordvinov & G. Fedyukovich (2017): *Synchronizing Constrained Horn Clauses*. In T. Eiter & D. Sands, editors: *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017, EPiC Series in Computing* 46, EasyChair, pp. 338–355, doi:`10.29007/gr5c`.

[20] L. M. de Moura & N. Bjørner (2008): *Z3: An Efficient SMT Solver*. In C.R. Ramakrishnan & J. Rehof, editors: *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, Springer, pp. 337–340, doi:`10.1007/978-3-540-78800-3_24`.

[21] A. Pettorossi & M. Proietti (2000): *Perfect Model Checking via Unfold/Fold Transformations*. In J. W. Lloyd, editor: *Proceedings of the First International Conference on Computational Logic (CL 2000), London, UK, 24-28 July*, Lecture Notes in Artificial Intelligence 1861, Springer-Verlag, pp. 613–628, doi:`10.1007/3-540-44957-4_41`.

[22] A. Reynolds & V. Kuncak (2015): *Induction for SMT Solvers*. In Deepak D'Souza, Akash Lal & Kim Guldstrand Larsen, editors: *Verification, Model Checking, and Abstract Interpretation - Proceedings of the 16th International Conference, VMCAI 2015, Mumbai, India,*, Lecture Notes in Computer Science 8931, Springer, pp. 80–98, doi:`10.1007/978-3-662-46081-8_5`.

[23] T. Sato & H. Tamaki (1984): *Transformational Logic Program Synthesis*. In: *Proceedings of the International Conference on Fifth Generation Computer Systems*, ICOT, pp. 195–201.

[24] H. Tamaki & T. Sato (1984): *Unfold/Fold Transformation of Logic Programs*. In S.-Å. Tärnlund, editor: *Proceedings of the Second International Conference on Logic Programming, ICLP '84*, Uppsala University, Uppsala, Sweden, pp. 127–138.

[25] H. Unno, S. Torii & H. Sakamoto (2017): *Automating Induction for Solving Horn Clauses*. In Rupak Majumdar & Viktor Kuncak, editors: *Proc. Computer Aided Verification - 29th Intern. Conf. CAV 2017, Heidelberg, Germany, Part II*, Lecture Notes in Computer Science 10427, Springer, pp. 571–591, doi:`10.1007/978-3-319-63390-9_30`.