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.

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) [2], (ii) Craig interpolation [15], and (iii) Property Directed Reachability (PDR) [1, 11]. Moreover, a variety of tools for satisfiability proofs, called *CHC solvers*, has been made available to the scientific community. Let us mention: Eldarica [12], HSF [10], RAHFT [13], VeriMAP [4], and Z3 [16]. 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 becomes much harder, or even impossible, because those solvers do not usually incorporate induction principles relative to the data structures in use.

To avoid this difficulty, two approaches have recently been suggested. The first one consists in the incorporation into the CHC solvers of suitable induction principles [17, 19], and the second one consists

Submitted to:
Verification and Program Transformation (VPT 2019)

in the use of a CHC transformation strategy, based on the familiar fold/unfold rules [9, 18], whose goal is to generate an equisatisfiable set of CHCs where the inductively defined data structures are removed.

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 [8], which implements a transformation strategy for removing inductively defined data structures. Thus, if the clauses derived by the Elimination Algorithm have all their constraints in the LIA or Bool theory, then no modifications of the CHC solvers are needed to perform the required satisfiability proofs.

As usual in the transformation-based approach, the success of the Elimination Algorithm depends on the introduction of suitable predicate definitions.

The novel contribution of this paper is a technique, presented through an example, for introducing those suitable predicates, which we call *difference predicates*, because they express the relation between the values computed by two different functions. The definition of those predicates also corresponds to the statement of the lemmata which should be proved if one were to show the properties of interest by structural induction. Our example shows that, by extending the Elimination Algorithm with the introduction of difference predicates, we can remove inductively defined data structures in cases where the plain Elimination Algorithm would not terminate.

In Sections 2 and 3, we will present the verification of a property of a functional program that acts on lists of integers, by first (i) deriving by transformation, using 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).

2 Horn Clause Satisfiability for Program Verification

Let us consider the following functional program *InsertionSort*, which we write using the OCaml syntax [14]:

```
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.

Suppose 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:

$\forall$ `l. sumlist l` $=$ `sumlist (insertionSort l)` (Property *Sum*)

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 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:

$\forall$ `x,l. sumlist (ins x l)` $=$ `x` $+$ `(sumlist l)` (Lemma *L*)

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 [8, 19]. In our example, 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, `sumlist(L,M)`, `insertionSort(L,SL)`, and `ins(X,L,L1)` hold iff `sumlist L` $=$ `M`, `insertionSort L` $=$ `SL`, and `ins X L` $=$ `L1`, respectively, hold in program *InsertionSort*.

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

$\forall$ `l,m,sl,n. sumlist l` $=$ `m` $\wedge$ `insertionSort l` $=$ `sl` $\wedge$ `sumlist sl` $=$ `n` $\rightarrow$ `m` $=$ `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.

Moreover, starting from clauses `1`–`8`, the Elimination Algorithm [8] 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 Transformation of Constrained Horn Clauses

In this section we show that, if we extend the Elimination Algorithm [8] by a technique for introducing difference predicates, we are able to transform clauses `1`–`8` into a set of clauses without lists and 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 *define*, *fold*, *unfold*, and *replace* for CHCs [9, 18]. The details can be found in the paper where the algorithm was originally presented [8].

[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.

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 set *Gs* of input goals. *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 statement. *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 their arguments of basic types only.

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

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

We start off the transformation of clauses 1–8 so to get clauses without list variables by applying the Elimination Algorithm $\mathscr{E}$ to $Cls = \{\text{clause 2}, \ldots, \text{clause 8}\}$ and $Gs = \{\text{clause 1}\}$. 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 body of the clauses identify the various 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, 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 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. 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 manage to match the conjunction (9a.1, 9a.2) with the conjunction (12.1, 12.2) by the renaming substitution $\sigma = \{\mathtt{La}/\mathtt{T}, \mathtt{Ma}/\mathtt{M}, \mathtt{SLa}/\mathtt{ST},\}$, but we cannot extend this matching to the remaining atom 9a.3 because the substitution $\{\mathtt{SLa}/\mathtt{SU}, \mathtt{Na}/\mathtt{N1}\}$ 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 *non-matching conjunction* 9m.3 to its right (in general, also the non-matching 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 non-matching 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 *non-matching 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 the non-matching conjunction of clause 12 by the non-matching conjunction of clause 9m. This replacement can be done at the expense of adding to the body of clause 12 a new atom with a so-called *difference predicate*. This atom addition is required for preserving satisfiability of the derived clauses. In our case the difference predicate we introduce, called diff, is defined as follows:

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

Let us explain how this clause is constructed. Its body is made out of two conjunctions (separated by the || marker): (i) the first one is the non-matching conjunction of the clause to be folded (atoms 12.3 and 12.4 in our case), and (ii) the second one is the non-matching conjunction of the clause we will use for folding (atom 9m.3 in our case). The arguments H, Na, and N1 of the new predicate diff are the non-list variables occurring in the clause body we have now constructed (obviously these arguments can be placed in any order). This choice of the arguments of diff is in accordance with our goal of eliminating lists.

Note that the marker || separates the atoms to the left that are the atoms to be removed from the body of clause 12 from the atoms to its right that are the atoms to be added to the body of clause 12 so that folding may be performed (see Step 6 below).

The reader may note that the difference predicate diff(H,Na,N1) expresses the relation between the non-list output variable N1 of the atoms 12.3 and 12.4 that are removed and the non-list output

variable Na of the atom 9m.3 that is added (the input and 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). Indeed, clause 13 can be read (in functional notation) as follows:
if sumlist (ins H ST) = N1 and sumlist ST = Na in program *InsertionSort*, *then* the relation between Na and N1 is given by diff(H,Na,N1).

• *Step* 5. *Replace*. In the clause to be folded (clause 12 in our case) we replace the non-matching conjunctions (atoms 12.3 and 12.4 in our case) by: (i) the non-matching conjunction of the clause we will use for folding (that is, atom 9m.3), and (ii) the head of the definition of diff. By doing so 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 by construction) and we get:

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

It can be shown that the above Steps 1–6 preserve satisfiability of clauses in the sense that the clauses after those steps are satisfiable iff so are the clauses before those steps. This satisfiability preservation is a needed requirement for being able to prove property *Sum* by performing a satisfiability check. The detailed proof of this fact is outside the scope of the present paper. In particular, regarding the correctness of the replacement of Step 5, it can be shown that:
if (H1) sumlist(L,N) and ins(X,S,S1) define total functional relations, *and*
(H2) diff(H,Na,N1) is a functional relation, that is, there is a function f such that
diff(H,Na,N1) implies f(H,Na)=N1,
then the replacement of clause 12 by clauses 12f and 13 produces an equisatisfiable set of clauses.

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

Note also that in the above procedure for introducing difference predicates no extra variable renaming of clauses is necessary besides those of Step 2.

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 $Cls = \{\text{clause 2}, \ldots, \text{clause 6}\}$ and $InCls = \{\text{clause 13}\}$, 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 proves that this set of clauses is satisfiable by computing the following model expressible in LIA [2]:

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 predicates 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 if `diff(H,Na,N1)` holds, then the function `f` such that `f(H,Na)=N1` is the familiar '+' function on integers.

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

• *Note on the Introduction of Difference Predicates and Lemma Generation.* 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 Z3, we exactly get the CHC translation of Lemma *L* needed for proving Property *Sum* by structural induction on lists. Thus, the introduction of the difference predicates can be viewed, at least in some cases, as a way of generating the lemmata required during proofs by structural induction.

4 Concluding Remarks

Let us briefly discuss how the correctness and mechanization of the transformation technique presented through an example in this paper can be obtained for large classes of clauses.

The main hypothesis needed to show the correctness of our transformation technique is that the predicates occurring in the initial set of clauses define total functional relations. This property is guaranteed by construction whenever those predicates are the CHC translation of functional programs that terminate for all inputs. One more hypothesis that is needed is the functionality of the difference predicates introduced during the transformation. This functionality requirement can be checked in the model computed by the CHC solver, which is expressed as a set of LIA constraints.

In order to mechanize our transformation technique, we need to extend the Elimination Algorithm [8] with a suitable automated mechanism for introducing difference predicates. As shown in Section 3, this mechanism can be based on matching the clauses obtained by unfolding (clause 12, in our example) against the predicate definitions introduced in previous transformation steps (clause 9, in our example), and computing the "difference" between their bodies. 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 [3, 13]. We have made initial steps towards an implementation of such an extended Elimination Algorithm using the VeriMAP transformation and verification system [4].

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 example 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 [3, 5, 6, 8, 7, 13] 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.

[2] By using '`z3_4.8.4 -smt2 sumlist.transf.smt fp.engine=spacer dump_models=true`'.

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 our approach pays off in practice.

5 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] 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. Available at https://doi.org/10.1007/978-3-642-18275-4_7.

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

[3] 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. Selected and extended papers from Partial Evaluation and Program Manipulation 2013.

[4] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2014): *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, Springer, pp. 568–574. Available at: http://www.map.uniroma2.it/VeriMAP.

[5] 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.

[6] 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:http://dx.doi.org/10.1016/j.scico.2016.11.002. Available at http://www.sciencedirect.com/science/article/pii/S016764231630199X. Selected and Extended papers from the International Symposium on Principles and Practice of Declarative Programming 2015.

[7] 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. Available at https://doi.org/10.1017/S1471068417000497.

[8] 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. Available at https://doi.org/10.1017/S1471068418000157.

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

[10] 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.

[11] 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. Available at http://dx.doi.org/10.1007/978-3-642-31612-8_13.

[12] 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.

[13] B. Kafle, J. P. Gallagher & J. F. Morales (2016): *RAHFT: A Tool for Verifying Horn Clauses Using Abstract Interpretation and Finite Tree Automata*. In: *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.

[14] 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.

[15] 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.

[16] L. M. de Moura & N. Bjørner (2008): *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, Springer, pp. 337–340.

[17] 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. Available at https://doi.org/10.1007/978-3-662-46081-8_5.

[18] 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.

[19] 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. Available at https://doi.org/10.1007/978-3-319-63390-9_30.