Contract Verification using Catamorphisms and Constrained Horn Clause Transformations

Emanuele De Angelis
CNR-IASI – emanuele.deangelis@iasi.cnr.it

Fabio Fioravanti
DEc, University of Chieti-Pescara – fabio.fioravanti@unich.it

Alberto Pettorossi
DICII, University of Rome "Tor Vergata" – adp@iasi.cnr.it

Maurizio Proietti
CNR-IASI – maurizio.proietti@iasi.cnr.it

(Extended Abstract)

1 Introduction

In many program verification techniques (à la Floyd [9] and Hoare [12]) program meaning is specified by *contracts*, that is pairs of *precondition* and *postcondition* formulas. A program is said to be *partially correct* with respect to a given contract if the precondition holds before program execution, and the program terminates, then the postcondition holds. Many programming languages (e.g., Ada [3], Ciao [11], Eiffel [19], Scala [21], and Solidity [24]) provide support for contract specification.

Programmers use contracts to specify invariant properties of entire programs or program fragments (such as functions, methods, and loops) and these contracts are used by verifiers (e.g., BOOGIE [1], LEON [25], WHY3 [8], DAFNY [17], and STAINLESS [10]) to generate suitable verification conditions, i.e., formulas whose validity guarantees program correctness.

Verification conditions are usually checked by using theorem provers or, more often, Satisfiability Modulo Theory (SMT) solvers [2, 4, 13, 20] and Constrained Horn Clause (CHC) solvers, such as Eldarica [13] and SPACER [15], that support a wide range of logical theories, from basic data types, such as Integers and Booleans, to non-basic data types, such as Lists and Trees.

In the case of programs manipulating complex data structures, such as Algebraic Data Types (ADTs), the loop invariants may be quite complicated, and their verification may need the extension of solvers with extra machinery, including inductive proof rules [23, 26, 27], tree automata-based techniques [16], and suitable CHC abstractions [14].

We follow an approach to the verification of programs that manipulate ADTs which is based on the translation of the contract verification problem into an equivalent satisfiability problem for CHCs. Then, we build upon previous methods [5, 6] for transforming a given set of clauses into a new set of clauses where all ADT terms are removed. These methods are sound, that is, the satisfiability of the transformed clauses implies the satisfiability of the original set of clauses, and, under suitable hypotheses, it is complete, that is, the original and the transformed set of clauses are equisatisfiable [5, 7]. In this way, we separate 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.

The transformational approach compares well with induction-based solvers in many cases [5, 7], but it still suffers from problems similar to the ones faced when automating mathematical induction proofs. In particular, during ADT removal, the transformation process is not guaranteed to terminate if the lemmas which need be discovered, are not found.

Submitted to:
HCVS 2022

In this extended abstract we focus on contracts that are specified by means of *catamorphisms* [18] (actually, an extended notion of them), that is, functions defined by a simple recursion schema on the ADTs manipulated by the program. In Figure 1 we present the catamorphism schema `h` acting on lists. In that figure, `f` is assumed to be a catamorphism defined by an instance of that same schema, (ii) the second arguments of `h` and `f` are of sort list, while all other arguments are of basic sort, and (iii) the predicate `c` defines a total function from its first four arguments to its last one.

```
h(X,[],Res) :- Res=b.
h(X,[H|T],Res) :- f(X,T,Rf), h(X,T,R), c(X,H,Rf,R,Res).
```

Figure 1: The list catamorphism schema `h` in CHC syntax.

Examples of catamorphisms are functions that compute list length or tree size, and functions that check sortedness properties for lists or for various kinds of trees such as binary search trees and heaps. Catamorphisms allow us to define a significant class of program properties, and indeed have been used in some approaches that combine verification procedures with specific algorithms for checking satisfiability in ADT theories [22, 25]. The interest in this kind of properties is shown also by a recent paper [14] that presents a technique for solving CHCs defined on ADTs and catamorphisms.

We have developed a transformation algorithm that is guaranteed to terminate when contracts are specified by catamorphisms, and produces clauses acting on basic types only, whose satisfiability implies the satisfiability of the original set of clauses. The satisfiability of the transformed clauses is then checked by using CHC solvers such as Eldarica and SPACER, without using the theory of the ADTs at hand.

2 Verifying catamorphism-based contracts using CHC transformation

We present our technique by means of an example. Let us consider the functional Scala program *Reverse* for computing the reversal of a list depicted in Figure 2, where function preconditions and postconditions are specified by `require` and `ensuring` assertions, respectively. Here and in the following, some code fragments are not shown, for reasons of space. The contract for the function `rev` states that, if a list `l` of integers is sorted in *ascending* order (w.r.t. $\leq$) then the list `rev(l)` is sorted in *descending* order (w.r.t. $\geq$). The ascending (or descending) order for the list `l` is checked by the function `is_asorted(l)` (or `is_dsorted(l)`, respectively). The contract for the function `snoc(l,x)`, which appends the element `x` to the end of the list `l`, states that if the list `l` is in descending order and `leq_all(x,l)` holds (that is `x` is less than or equal to every element of `l`), then also `snoc(l,x)` is in descending order. STAINLESS, which is a verifier for Scala programs [10], is not able to check the validity of the contract for `rev`, because it fails to establish the precondition for the function call `snoc(rev(xs),x)` (see the `Cons` case for `rev`).

The translation of functions to CHC predicates is based on the *call-by-value* semantics and we have that the set *ReverseCHCs* of clauses that encode program *Reverse* is satisfiable if and only if the contracts for `rev` and `snoc` are valid (see Figure 3). For example, function `rev` is translated to a predicate `rev` such that `rev(X,Y)` holds iff the function `rev(X)` evaluates to `Y`. In order to prove that a contract for a given function `f` is valid, we need to prove that $\forall$`x. precond(x)` $\rightarrow$ `postcond(x,f(x))`. In Figure 3, contracts for the functions `rev` and `snoc` are encoded by the constrained goals `GR` and `GS` (that is, clauses which have the head `false`), respectively.

Our transformation algorithm T_{Cata}, not shown in this extended abstract, works by introducing new predicate symbols with basic types only, corresponding to existentially quantified conjunctions of a predicate translating a program function (such as `rev`) and some catamorphism predicates (such as `is_asorted`). If contracts are indeed specified using catamorphisms, such as those used in our example,

```
def rev(l: List[Int]): List[Int] = {
    require(is_asorted(l))
    l match {case Nil() => Nil()
             case Cons(x,xs) => snoc(rev(xs),x)
    } } ensuring{ res => is_dsorted(res) }
def snoc(l: List[Int], x:Int): List[Int] = {
    require(is_dsorted(l) && leq_all(x,l))
    l match {case Nil() => Cons(x,Nil())
             case Cons(y,ys) => Cons(y,snoc(ys,x))
    } } ensuring{ res => is_dsorted(res) }
def leq_all(x: Int, l: List[Int]): Boolean = {
    l match { case Nil() => true
              case Cons(y, ys) => if (x > y) {false} else {leq_all(x, ys)} } }
def is_asorted(l: List[Int]): Boolean = {
    l match {case Nil() => true
             case Cons(x,xs) => is_empty(xs) || (x =< hd(xs) && is_asorted(xs)) } }
def is_dsorted(l: List[Int]): Boolean = {
    l match {case Nil() => true
             case Cons(x,xs) => is_empty(xs) || (x >= hd(xs) && is_dsorted(xs)) } }
```

Figure 2: Program *Reverse* with the contracts for the functions `rev` and `snoc`.

```
    rev([],[]).
    rev([H|T],R) :- rev(T,S), snoc(S,H,R).
    snoc([],X,[X]).
    snoc([X|Xs],Y,[X|Zs]) :- snoc(Xs,Y,Zs).
    is_asorted([],Res) :- Res.
    is_asorted([X|Xs],Res) :- Res = (IsDefXs => (X=<HdXs & ResXs)),
                              hd(Xs,IsDefXs,HdXs), is_asorted(Xs,ResXs).
    is_dsorted([],Res) :- Res.
    is_dsorted([X|Xs],Res) :- Res = (IsDefXs => (X>=HdXs & ResXs)),
                              hd(Xs,IsDefXs,HdXs), is_dsorted(Xs,ResXs).
    hd([],IsDef,Hd) :- ~IsDef & Hd=0.
    hd([H|T],IsDef,Hd) :- IsDef & Hd=H.
    leq_all(N,[],B) :- B.
    leq_all(N,[X|Xs],B) :- leq_all(N,Xs,B1), B = (N=<X & B1).
GR. false :- (BL & ~BR), rev(L,R), is_asorted(L,BL), is_dsorted(R,BR).
GS. false :- (BX & BA & ~BC), snoc(A,X,C), is_dsorted(A,BA),
               leq_all(X,A,BX), is_dsorted(C,BC).
```

Figure 3: The set *ReverseCHCs* of clauses for the program *Reverse*. In constraint formulas we use integer and boolean variables, the predicate '=' (equality) and the operators '~' (negation), '&' (conjunction), and '=>' (implication). The predicates `is_asorted`, `is_dsorted`, `hd`, and `leq_all` are catamorphisms.

T_{Cata} always terminates and produces clauses where only new predicate symbols occur. For instance, T_{Cata} introduces the following predicate for the contract for `rev` (goal `GR`):

```
new3(BR,N,B,IsDef,Hd,BL) :- is_asorted(L,BL), hd(L,IsDef,Hd), rev(L,R),
                             is_dsorted(R,BR), leq_all(N,R,B).
```

and derives the following clauses:

```
new7(A,B,C,D,E,F,G,H,D,I,J) :- A & B=D & C=(K=>((D>=L)&M)) & E & ~F & G=0 & H &
                       & J=((I=<D)&N) & M & ~K & L=0 & N.
new7(A,B,C,D,E,F,G,H,D,I,J) :- (A & B=K & C=(L=>((K>=M)&N)) & E=((D=<K)&T) & F &
                       & G=K & H=(P=>((K>=Q)&R)) & J=((I=<K)&S) & (R&T)=>N),
                       new7(L,M,N,D,T,P,Q,R,D,I,S).
```

```
new3(A,B,C,D,E,F) :- A & C & ~D & E=0 & F.
new3(A,B,C,D,E,F) :- (D & E=G & F=(H=>((G=<I)&J)) & J=>K & (K&L)=>A),
                      new3(K,G,L,H,I,J), new7(M,N,A,G,L,T,P,K,G,B,C).
false :- (A & ~B), new3(B,C,D,E,F,A).
```

State-of-the-art CHC solvers, such as Eldarica and SPACER, fail to prove the satisfiability of *ReverseCHCs* but are able prove the satisfiability of the transformed, ADT-free clauses.

The results of the experiments we performed using a prototype implementation of the transformation algorithm T_{Cata}[1], show that our approach is effective on a large (about 190), significant class of properties, among which there are properties of programs that sort lists and perform various tree manipulations, such as tree insertion, deletion, and traversal. Similarly to our *Reverse* example, in most verification problems of our benchmark, Eldarica and SPACER fail to terminate on the original clauses with ADTs while they prove satisfiability of the transformed clauses.

Currently, we are refining our tool and, in the near future, we plan to conduct a more extensive experimental comparison with other solvers for CHCs on ADTs, and in particular ADTIND [27], RACER [14], and REGINV [16], and also with program verifiers like the already mentioned DAFNY, STAINLESS, and WHY3. For that purpose, we need to develop automatic translators between benchmarks, which at the moment are not available for some of those tools.

References

[1] M. Barnett, B.-Y. E. Chang, R. De Line, B. Jacobs & K. R. M. Leino (2006): *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*. In: *Formal Methods for Components and Objects*, LNCS 4111, Springer, pp. 364–387.

[2] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds & C. Tinelli (2011): *CVC4*. In: *23rd CAV '11*, LNCS 6806, Springer, pp. 171–177, doi:10.1007/978-3-642-22110-1_14.

[3] G. Booch & D. Bryan (1994): *Software Engineering with Ada (3rd ed.)*. Benjamin/Cummings series in Object-Oriented Software Engineering, Benjamin/Cummings.

[4] A. Cimatti, A. Griggio, B. Schaafsma & R. Sebastiani (2013): *The MathSAT5 SMT solver*. In: *19th TACAS '13*, LNCS 7795, Springer, pp. 93–107.

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

[6] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2020): *Removing Algebraic Data Types from Constrained Horn Clauses Using Difference Predicates*. In: *IJCAR 2020*, LNAI 12166, Springer, pp. 83–102, doi:10.1007/978-3-030-51074-9_6.

[7] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2022): *Satisfiability of Constrained Horn Clauses on Algebraic Data Types: A Transformation-based Approach*. *Journal of Logic and Computation*, doi:10.1093/logcom/exab090.

[8] J.-C. Filliâtre & A. Paskevich (2013): *Why3 - Where Programs Meet Provers*. In: *Programming Languages and Systems, ESOP '13*, LNCS 7792, Springer, pp. 125–128.

[9] R. W. Floyd (1967): *Assigning Meanings to Programs*. In: *Proceedings of Symposium on Applied Mathematics, Vol. 19*, American Mathematical Society, Providence, R.I., USA, pp. 19–32.

[10] J. Hamza, N. Voirol & V. Kuncak (2019): *System FR: Formalized Foundations for the Stainless Verifier*. *Proc. ACM Program. Lang.* 3(OOPSLA), pp. 166:1–166:30, doi:10.1145/3360592.

[1] `https://tinyurl.com/catamorphisms`

[11] M. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales & G. Puebla (2012): *An Overview of Ciao and its Design Philosophy*. *Theor. Pract. Logic Program.* 12(1–2), pp. 219–252, doi:10.1017/S1471068411000457.

[12] C.A.R. Hoare (1969): *An Axiomatic Basis for Computer Programming*. *CACM* 12(10), pp. 576–580, 583, doi:10.1145/363235.363259.

[13] H. Hojjat & Ph. Rümmer (2018): *The ELDARICA Horn Solver*. In: *Formal Methods in Computer Aided Design, FMCAD 2018*, IEEE, pp. 1–7, doi:10.23919/FMCAD.2018.8603013.

[14] H. Govind V. K., Sh. Shoham & A. Gurfinkel (2022): *Solving Constrained Horn Clauses Modulo Algebraic Data Types and Recursive Functions*. *Proc. ACM Program. Lang.* 6 (POPL), pp. 1–29, doi:10.1145/3498722.

[15] A. Komuravelli, A. Gurfinkel & S. Chaki (2014): *SMT-Based Model Checking for Recursive Programs*. In: *26th CAV '14*, LNCS 8559, Springer, pp. 17–34.

[16] Y. Kostyukov, D. Mordvinov & G. Fedyukovich (2021): *Beyond the Elementary Representations of Program Invariants over Algebraic Data Types*. In: *PLDI '21*, ACM, pp. 451–465, doi:10.1145/3453483.3454055.

[17] K. R. M. Leino (2013): *Developing Verified Programs with Dafny*. In: *Intl. Conf. on Software Engineering '13*, IEEE Press, pp. 1488–1490, doi:10.1109/ICSE.2013.6606754.

[18] E. Meijer, M. M. Fokkinga & R. Paterson (1991): *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*. In: *Functional Programming Languages and Computer Architecture, 5th ACM Conference*, LNCS 523, Springer, pp. 124–144, doi:10.1007/3540543961_7.

[19] B. Meyer (1991): *Eiffel: The Language*. Prentice-Hall.

[20] L. M. de Moura & N. Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *14th TACAS '08*, LNCS 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[21] M. Odersky, L. Spoon & B. Venners (2011): *Programming in Scala: A Comprehensive Step-by-Step Guide*, 2nd edition. Artima Incorporation, Sunnyvale, CA, USA.

[22] T.-H. Pham, A. Gacek & M. W. Whalen (2016): *Reasoning About Algebraic Data Types with Abstractions*. *J. Autom. Reason.* 57(4), pp. 281–318, doi:10.1007/s10817-016-9368-2.

[23] A. Reynolds & V. Kuncak (2015): *Induction for SMT Solvers*. In: *16th VMCAI*, LNCS 8931, Springer, pp. 80–98, doi:10.1007/978-3-662-46081-8_5.

[24] Solidity (2022): *Solidity v 0.8.12 Documentation*. `https://docs.soliditylang.org/`.

[25] Ph. Suter, A. S. Köksal & V. Kuncak (2011): *Satisfiability Modulo Recursive Programs*. In: *18th SAS '11*, LNCS 6887, Springer, pp. 298–315.

[26] H. Unno, S. Torii & H. Sakamoto (2017): *Automating Induction for Solving Horn Clauses*. In: *29th CAV '17, Part II*, LNCS 10427, Springer, pp. 571–591, doi:10.1007/978-3-319-63390-9_30.

[27] W. Yang, G. Fedyukovich & A. Gupta (2019): *Lemma Synthesis for Automating Induction over Algebraic Data Types*. In: *CP 2019*, LNCS 11802, Springer, pp. 600–617.