Contract Strengthening through Constrained Horn Clause Verification

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

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

Alberto Pettorossi
DICII, University of Rome 'Tor Vergata', Italy
pettorossi@info.uniroma2.it

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

The functional properties of a program are often specified by providing a contract for each of its functions. A contract of a function consists of a pair of formulas, called a precondition and a postcondition, which, respectively, should hold before and after execution of that function. It might be the case that the contracts supplied by the programmer are not adequate to allow a verification system to prove program correctness, that is, to show that for every function, if the precondition holds and the execution of the function terminates, then the postcondition holds. We address this problem by providing a technique which may strengthen the postconditions of the functions, thereby improving the ability of the verifier to show program correctness. Our technique consists of four steps. First, the translation of the given program, which may manipulate algebraic data structures (ADTs), and its contracts into a set of constrained Horn clauses (CHCs) whose satisfiability implies the validity of the given contracts. Then, the derivation, via CHC transformation performed by the VeriCaT tool, of a new set of CHCs that manipulate only basic sorts (such as booleans or integers) and whose satisfiability implies the satisfiability of the original set of clauses. Then, the construction of a model, if any, of the new, derived CHCs using the CHC solver SPACER for basic sorts. Finally, the translation of that model into the formulas that suitably strengthen the postconditions of the given contracts. We will present our technique through an example consisting of a Scala program for reversing lists. Note that the STAINLESS verifier is not able to prove the correctness of that program when considering the given contracts, while it succeeds when considering the contracts with the strengthened postconditions constructed by applying our technique.

1 Introduction

In many program verification techniques à la Floyd-Hoare [11, 16], the meaning of a program is specified by providing a *contract*, that is, a pair of a *precondition* and a *postcondition* formula for each of the program functions. A program function is said to be *partially correct* with respect to a given contract if the precondition holds before function execution, and the function terminates, then the postcondition holds. A program function is said to be *totally correct* if it is partially correct and it terminates whenever its precondition holds. Many programming languages (for instance, Ada [3], Ciao [15], Eiffel [22], Scala [24], and Solidity [26]) provide support for contract specification.

Programmers write contracts to specify invariant properties of entire programs or program fragments (such as functions, methods, and loops) and these contracts may be used by verifiers (e.g., BOOGIE [1], LEON [27], WHY3 [10], DAFNY [20], and STAINLESS [14]) to generate and possibly prove suitable verification conditions, that is, formulas whose validity guarantees program correctness.

G.W. Hamilton, T. Kahsai, M. Proietti (Eds.):
Joint International Workshops on
Horn Clauses for Verification and Synthesis (HCVS 2022)
Verification and Program Transformation (VPT 2022)
EPTCS 373, 2022, pp. 23–34, doi:10.4204/EPTCS.373.3

Verification conditions are usually proved by using theorem provers or Satisfiability Modulo Theory (SMT) solvers [2, 4, 17, 23] and constrained Horn clause (CHC) solvers, such as Eldarica [17] and SPACER [18]. In these solvers there is support for a wide range of logical theories, including basic data types, such as integers and booleans, and also more complex data structures.

In the case of programs manipulating complex data structures, such as Algebraic Data Types (ADTs), loop invariants and contracts may be quite complicated and their verification may require the enhancement of the solvers by incorporating inductive proof rules [25, 28, 29], or tree automata-based techniques [19], or CHC abstractions [12].

Besides the presence of ADTs, there is often one additional reason that makes it difficult to prove the validity of the contracts when using an automated verification system. It is the fact that the contracts specified by the programmer are not sufficiently detailed in specifying the behaviour of the functions.

In order to clarify this point, let us consider the case of a program made out of some functions which may call each other, possibly in a mutually recursive way. Each of these functions has its own contract. Now it may happen that a program verifier is able to prove the contract of a particular function, say g, while it fails to prove the contract of another function, say f, because f calls g and the contract that has been proved for g is not detailed enough. This may happen because the programmer, when specifying contracts, did not take into account the fact that g is called by f, and for the proof of the contract of f a stronger, more detailed contract for g is indeed required. Moreover, these contract interdependencies are not always easy to take into account by the programmer, and this happens in particular when the program is made out of many function definitions.

In this paper we present a novel technique for strengthening the function contracts in the case of programs that manipulate ADTs, and then, our technique allows an easier proof of contract correctness. Our technique consists of four steps.

Step (i): First, we translate the given program with its contracts into a set of CHCs, so that the contract verification problem is translated into an equivalent satisfiability problem for that set of CHCs.

Step (ii): Then, by using already known methods [6, 7], we transform this set of CHCs into a new set where all ADT terms are removed. In particular, we use the VeriCaT tool [9]. These methods are sound, in the sense that the satisfiability of the transformed clauses implies the satisfiability of the original set of clauses. Under suitable hypotheses, these methods are also complete, that is, the original and the transformed set of clauses are equisatisfiable [8]. 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. Usually, during this transformation new predicate symbols are introduced in the derived CHCs.

Step (iii): Then, we invoke a CHC solver (in our case SPACER) to show the satisfiability of the new, transformed set of CHCs and to construct the models, if any, of the new predicates which have been introduced.

Step (iv): Finally, from those models we construct the formulas which strengthen the given contracts.

The viability and the power of this novel technique will be shown in the following sections through a simple example dealing with lists.

2 Verification of Program Contracts

In this section we present the program verification problem we consider with the help of an example. Let us consider the Scala program *Reverse*, depicted in Figure 1, for computing the reversal of a list. In that program the function preconditions and postconditions are specified by `require` and `ensuring` assertions,

respectively. 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 (that is, it is sorted w.r.t. $\geq$). The ascending (or descending) order for 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 element `x` to the end of list `l`, states that if list `l` is sorted in descending order and `leq_all(x,l)` holds (that is, element `x` is less than or equal to every element of `l`), then also `snoc(l,x)` is sorted in descending order. In program *Reverse* we also need the function `hd` which, given a list of integers, returns a pair consisting of a boolean and an integer. If the given list is not empty and `h` is its head, then the returned pair is $\langle$`true`, `h`$\rangle$, while it is $\langle$`false`, `0`$\rangle$, if the list is empty, being `0` an arbitrary integer value (which is never used elsewhere in the program).

```
object Reverse {
  def rev(l: List[BigInt]): List[BigInt] = {
    require(is_asorted(l))                              // precondition of rev
    l match {
      case Nil() => Nil[BigInt]()
      case Cons(x, xs) => snoc(rev(xs),x) }
      } ensuring { res => is_dsorted(res) }             // postcondition of rev

  def snoc(l: List[BigInt], x:BigInt): List[BigInt] = {
    require(is_dsorted(l) && leq_all(x,l))              // precondition of snoc
    l match {
      case Nil() => Cons(x,Nil())
      case Cons(y, ys) => Cons(y,snoc(ys,x)) }
         } ensuring { res => is_dsorted(res) }          // postcondition of snoc

  def is_asorted(l: List[BigInt]): Boolean = {
    l match {
      case Nil() => true
      case Cons(x,xs) => !(hd(xs)._1) || (x <= (hd(xs)._2) && is_asorted(xs))  } }

  def is_dsorted(l: List[BigInt]): Boolean = {
      l match {
      case Nil() => true
      case Cons(x,xs) => !(hd(xs)._1) || (x >= (hd(xs)._2) && is_dsorted(xs))  } }

  def hd(l: List[BigInt]): (Boolean, BigInt) = {
    l match {
      case Nil() => (false, BigInt(0))
      case Cons(x, xs) => (true, x) } }

  def leq_all(x: BigInt, l: List[BigInt]): Boolean = {
    l match {
      case Nil() => true
      case Cons(y, ys) => if (x > y) {false} else { leq_all(x, ys) } } }
}  // end of object Reverse
```

Figure 1: Program *Reverse* with the contracts for the functions `rev` and `snoc`. '`!`', '`&&`', and '`||`' denote boolean negation, conjunction, and disjunction, respectively. '`p._1`', and '`p._2`' denote the first and the second projection of a given pair `p`, respectively.

In order to prove the validity of a contract $\langle$`precond(x), postcond(x,f(x))`$\rangle$ for a given function `f`, we need to prove that $\forall$`x. precond(x)` $\rightarrow$ `postcond(x,f(x))`. Now, if we submit to STAINLESS [14], which is a verifier for Scala programs, the above *Reverse* program with the initial directives:

```
import stainless.proof._
import stainless.lang._
import stainless.collection._
```

we get that STAINLESS is not able to check the validity of the contract for `rev`, because it fails to establish (within the timeout of 100 s) the precondition for the function call '`snoc(rev(xs),x)`' (occurring in the `Cons` case for `rev`), which is needed to use the postcondition of `snoc` and prove that `rev(Cons(x, xs))` is sorted in descending order. Indeed, STAINLESS returns the following warning:

```
=> TIMEOUT case Cons(x, xs) => snoc(rev(xs),x)
```

In the next section we will see in action our technique for strengthening contracts in the case of the *Reverse* example. By using the strengthened contract for the function `rev`, STAINLESS is indeed able to construct, as desired, a proof of the validity of both contracts of that example, that is, the validity of the contracts for the functions `rev` and `snoc`.

3 Verifying Contracts via CHC Satisfiability

As already mentioned at the end of Section 1 our technique for strengthening the contracts is made out of four steps which we now perform in the case of the *Reverse* program shown in Figure 1.

Step (i). We translate the given Scala program and its contracts into the set *ReverseCHCs* of clauses shown in Figure 2. In this translation we maintain the *call-by-value* semantics of the Scala program in the sense that, if a function `f` applied to the input `X` evaluates to output `Y`, then the atom `f(X,Y)` occurs in the least model of the set *ReverseCHCs*.

Several techniques for translating imperative and functional programs have been defined in the literature [5, 13]. Even if no specific tool is available, we may assume that, by applying one of those techniques we get a translation of Scala functions to CHC predicates that guarantees that the set *ReverseCHCs* is satisfiable if and only if the contracts for the functions `rev` and `snoc` of the program *Reverse* are valid. In Figure 2, the contracts for the functions `rev` and `snoc` are encoded by the two constrained goals `GR` and `GS`, respectively. Here and in what follows, we call 'constrained goal' (or 'goal', for short) any clause whose head is `false`.

Note that when writing clauses, we often prefer writing the variable `X`, instead of the constraint `X=true`, and the negated variable `~X`, instead of the constraint `X=false`. In particular, in clause `GR` of Figure 2 we have written `(BL & ~BR)`, instead of the equivalent constraint `(BL=true & BR=false)`.

Now, in order to show the validity of the contracts for `rev` and `snoc`, we have to show that the set *ReverseCHCs* of clauses is satisfiable. Unfortunately, the CHC solvers Eldarica and SPACER are not capable to solve this satisfiability problem. This is basically due to the fact that those solvers lack any form of inductive reasoning on lists and, moreover, they do not use the information about the validity of the contract for `snoc` during the proof of satisfiability of the goal which encodes the contract for `rev`.

Then, we proceed according to Step (ii) of our technique, which consists in applying a transformation that removes all ADT terms from *ReverseCHCs*. Indeed, we apply Algorithm $\mathcal{T}_{cata}$ [9], implemented in the VeriCaT tool, and we get the new set *TransfReverseCHCs* of clauses (see Figure 3), whose satisfibility implies the satisfiability of the set *ReverseCHCs*. In particular, starting from goal `GR`, by transformation we obtain clauses `T1`–`T5`, and starting goal `GS`, by transformation we obtain clauses `T6`–`T8`.

Let us make a minor remark about the numbering of the new predicates occurring in clauses `T1`–`T5` obtained by clause transformations starting from goal `GR`. (A similar remark applies to clauses `T6`–`T8` when starting from goal `GS`.) VeriCaT uses for the new predicates to be introduced, a progressive numbering starting from the name `new1`, that is, it uses `new1`, `new2`, and so on. Now only predicates `new3` and `new7` occur in clauses `T1`–`T5` because of the following two reasons: (i) during transformation, some of the new predicates that are introduced, are generalizations of already introduced ones, and thus one can replace the less general predicates by the more general ones, and (ii) some atoms with new predicates

```
/* --------- CHC translation of 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).

/* --------- CHC translation of the functions used by the contracts  --------- */
    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) :- B = (N=<X & B1), leq_all(N,Xs,B1).

/* --------- CHC translation of the contracts of the functions rev and snoc --------- */
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 2: 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).

are unfolded during clause transformation, and thus they will not occur in the derived final clauses.

```
/* --------- T1-T5: clauses derived for the rev contract GR  --------- */
T1. 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.
T2. 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).
T3. new3(A,B,C,D,E,F) :- A & C & ~D & E=0 & F.
T4. 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).
T5. false :- A & ~B, new3(B,C,D,E,F,A).                        /* --- folded from clause GR */

/* --------- T6-T8: clauses derived for the snoc contract GS --------- */
T6. new2(A,B,C,D,E,F,G,H,I) :- D=I & I=J & A & B=J & C=(K=>(J>=L & M)) & E &
         & ~F & G=0 & H & M & ~K & L=0.
T7. new2(A,B,C,D,E,F,G,H,I) :- D=I & D=J & I=K & K=J & L=M & A & B=M & C=(N=>(M>=V & P)) &
         & E=(D=<L & Q) & F & G=L & H=(R=>(L>=S & T)) & (T & Q)=>P, new2(N,V,P,J,Q,R,S,T,K).
T8. false :- A=B & ~((C & D)=>E), new2(F,G,E,B,D,H,I,C,A).     /* --- folded from clause GS */
```

Figure 3: The set *TransfReverseCHCs* of clauses obtained by Algorithm $\mathcal{T}_{cata}$ [9] at the end of Step (ii).

It is not really important that the programmer understands the meaning of the newly introduced predicates, as they are obtained in a fully automatic way by an algorithm whose soundness is guaranteed in all cases. It is only important to note that all variables in *TransfReverseCHCs* are of sort boolean or integer. Indeed, Algorithm $\mathcal{T}_{cata}$ always terminates and generates a set of clauses without ADT variables in the case where, as in our set *ReverseCHCs* of clauses, the contracts are specified by means of *catamorphisms* [9], that is, total functions defined by a simple recursion schema on the ADT structure. (The notion of catamorphism used here is an adaptation to CHCs of the one popularized by Meijer et al. [21] in the area of functional programming.)

The kind of catamorphisms we have used in our example are all instances of the list catamorphism schema `h` for CHCs depicted in Figure 4. The recursion of predicate `h` is on its second argument, which has sort list. We leave to the reader to check that, indeed, the predicates `is_asorted`, `is_dsorted`, `hd`, and `leq_all` (see Figure 2) that we have used for specifying the contracts of `rev` and `snoc`, are all list catamorphisms. In particular, `leq_all` is a list catamorphism according to the schema of Figure 4 by taking `f(X,T,Rf)` to be the atom `true`, and `c(N,X,Rf,B1,B)` to be defined by the constraint '`B = (N=<X & B1)`'.

```
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 4: Clauses defining the list catamorphism schema `h`.

In Figure 4, we assume that: (i) `f` is a catamorphism defined by an instance of the same schema of that figure, (ii) the second arguments of `h` and `f` are of sort list, while all other arguments are of basic sort (either boolean or integer), and (iii) the predicate `c` defines a total function from its first four arguments to its last one.

As already mentioned, we have that if clauses `T1`–`T5` are satisfiable then the contract for `rev` is valid and, likewise, if clauses `T6`–`T8` are satisfiable then the contract for `snoc` is valid.

Having derived the set *TransfReverseCHCs* of clauses without ADT variables, we are ready to perform Step (iii) of our technique. Thus, we invoke a CHC solver (in our case SPACER) which, hopefully, is capable to prove the satisfiablity of *TransfReverseCHCs*, because in this set of clauses there are variables of basic sort only. Indeed, SPACER given clauses `T1`–`T8`, returns the answer '`sat`' stating that *TransfReverseCHCs* is satisfiable. At this point, having proved the satifiability of *TransfReverseCHCs*, we have proved the validity of the contracts for `rev` and `snoc`.

Now we know that the contracts for `rev` and `snoc` are valid, even if the contract for `rev` is not provable by STAINLESS. However, it is desirable to perform a further step (it is Step (iv) of our technique that we will describe in the next section) and derive strengthened contracts for `rev` and `snoc` whose validity can be automatically shown by STAINLESS, without appealing to an external verification system based on the first three steps of our technique. The need for those strengthened contracts comes from the software engineering requirement of having a single framework (Scala, in our case) where one specifies programs and contracts, and also proves contract validity (using STAINLESS, in our case). In the next section we will show how to comply with this requirement.

Note also that, with respect to the given contracts, the strengthened contracts provide a more detailed documentation of the program at hand, and they allow the programmer to better understand the correctness of the program, as the transformation-based proofs of the contracts are often hard to follow.

4 Strengthening Program Contracts using CHC Models

Now we show how to derive strengthened contracts for `rev` and `snoc` so that the STAINLESS verifier can prove their validity. This derivation can be done by:

(1) taking into account the definitions of the new predicates introduced by VeriCaT during the CHC transformation of Step (ii), and

(2) constructing the models of those new predicates as an outcome of the satisfiability proof of the CHCs performed by SPACER at Step (iii).

As we have mentioned at the end of Section 2, since STAINLESS is unable to show the contract for `rev`, while it is able to show the contract for `snoc`, we proceed by explaining how to get strengthened contracts

by considering only the models of clauses T1–T5 relative to the rev contract. Those clauses refer to the new predicates new3 and new7 which VeriCaT introduced during Step (ii) by the following definition clauses (modulo variable renaming):

```
D3. new3(BR,N,B,IsDef,Hd,BL) :- is_asorted(L,BL), leq_all(N,Res,B), hd(L,IsDef,Hd),
               rev(L,Res), is_dsorted(Res,BR).

D7. new7(A1,B1,BC,X,BE,F1,G1,BA,X,J1,K1) :- hd(L,F1,G1), hd(Res,A1,B1),
               is_dsorted(L,BA), leq_all(X,L,BE), snoc(L,X,Res), is_dsorted(Res,BC),
               leq_all(J1,Res,K1).
```

(Note that, in order to eliminate the ADT variables, in the above clauses D3 and D7, the arguments of their head atoms are the variables of basic sort occurring in their associated bodies.)

At Step (iii) SPACER shows the satisfiability of clauses T1–T5 by constructing models for new3 and new7. The model for new3(BR,N,B,IsDef,Hd,BL) is:

```
M3.  (~IsDef => (BR & B))  &  (BL => (BR & ((Hd>=N) => B)))          (model for new3)
```

and the model for new7(A1,B1,BC,X,BE,F1,G1,BA,X,J1,K1) is:

```
M7.  (BE & (X >= J1)) => K1.                                        (model for new7)
```

First, we consider the definition clause D3 and the model M3 for new3. From the left conjunct (~IsDef => (BR & B)) of the model we have that if the list l is empty, being IsDef=false (see the clauses for hd in Figure 2), then: (i) the reversed list res is empty, (ii) res is vacuously sorted in descending order, and (iii) leq_all(n,res) vacuously holds for all integers n. Hence, we get the following formula (using the Scala syntax), where the variable n has been universally quantified (indeed, it is neither an input nor an output variable of rev):

```
(3.1)   forall((n:BigInt) => (!(hd(l)._1) ==> (is_dsorted(res) && leq_all(n,res))))
```

From the second conjunct (BL => (BR & ((Hd>=N) => B))) of the model M3 for new3, we get the following formula (again the variable n has been universally quantified):

```
(3.2)   forall((n: BigInt) => (is_asorted(l) ==>
                        (is_dsorted(res) && ((hd(l)._2 >= n) ==> leq_all(n,res)))))
```

Now the conjunction of the given postconditon of the function rev, that is, is_dsorted(res) (see the definition of rev in Figure 1), and formulas (3.1) and (3.2) can be suitably simplified by taking into account the precondition of rev, that is, is_asorted(l). After that simplification, we eventually get the strengthened postcondition for rev shown in Figure 5.

```
... ensuring { res => is_dsorted(res) &&
               forall((n: BigInt) => ((!(hd(l)._1) ==> leq_all(n,res))  &&
                                      ((hd(l)._2 >= n) ==> leq_all(n,res)))) }
```

Figure 5: Strengthened postcondition for rev. Version 1.

Then, it remains to consider the definition clause D7 and the model M7 for new7. Now BE can be replaced by true, because BE occurs in the atom leq_all(X,L,BE) in the body of clause D7 and by the precondition of snoc (see Figure 1), we have that leq_all(x,l) holds. Moreover, in the two atoms snoc(L,X,Res) and leq_all(J1,Res,K1) occurring in the body of clause D7, the value of the variable Res is that of res of the Scala function snoc, and the value of the variable K1 is the result of the Scala function leq_all(j1,res). Hence, from the model for new7, we get the following formula to be used for strengthening the postcondition for snoc:

```
(7)   forall((j1: BigInt) => ((x >= j1)) ==> leq_all(j1,res)))
```

where the variable `j1` has been universally quantified, because it is neither an input nor an output variable of `snoc`. Thus, the conjunction of the given postcondition of `snoc`, that is, `is_dsorted(res)` (see the definition of `snoc` in Figure 1), and formula `(7)`, gives us the strengthened postcondition for `snoc` shown in Figure 6.

```
... ensuring { res => is_dsorted(res) &&
               forall((j1: BigInt) => ((x >= j1) ==> leq_all(j1,res))) }
```

Figure 6: Strengthened postcondition for `snoc`.

By using these new, strengthened postconditions of the contracts for `rev` and `snoc`, instead of the old postconditions, we have that STAINLESS is able to prove the correctness of both contracts, as desired.

Let us briefly discuss the soundness of our technique for strengthening contracts. We consider the class of contracts specified by catamorphisms considered in previous work [9], where the termination of the transformation algorithm $\mathcal{T}_{cata}$ is guaranteed. Soundness of our technique will be shown by proving that, if a given contract is valid and a CHC solver is able to prove the satisfiability of the set of clauses obtained by $\mathcal{T}_{cata}$, then also the strengthened version of the contract, which is derived by using our technique, is valid.

For reasons of simplicity, let us assume that we want to show the validity of a contract for the function $\mathtt{f} : \alpha \to \beta$, where α and β are ADTs. Since the contract is specified by catamorphisms, its validity can be expressed by a formula of the form:

$$\forall \mathtt{x},\mathtt{y},\mathtt{v}_1,\mathtt{v}_2.\ (\mathtt{f}(\mathtt{x})=\mathtt{y} \wedge \mathtt{p}_1(\mathtt{x})=\mathtt{v}_1 \wedge \mathtt{p}_2(\mathtt{y})=\mathtt{v}_2 \ \to\ \mathtt{c}(\mathtt{v}_1,\mathtt{v}_2)) \qquad (\mathtt{C})$$

where $\mathtt{p}_1$ and $\mathtt{p}_2$ are (tuples of) catamorphisms and $\mathtt{c}(\mathtt{v}_1,\mathtt{v}_2)$ is a constraint on integer and boolean variables. For instance, in our *Reverse* example, `C` is the formula

$$\forall \mathtt{l},\mathtt{res},\mathtt{b1},\mathtt{b2}.\ (\mathtt{rev}(\mathtt{l})=\mathtt{res} \wedge \mathtt{is\_asorted}(\mathtt{l})=\mathtt{b1} \wedge \mathtt{is\_dsorted}(\mathtt{res})=\mathtt{b2} \ \to\ (\mathtt{b1} \to \mathtt{b2}))$$

Let us assume that the contract is valid. Thus, if we consider: (i) the set P of CHCs that translates the program including the function `f`, and (ii) the following goal `G` that translates the contract (where, as in Figure 2, '`~`' denotes boolean negation):

`G: false :- ~c(V1,V2), f(X,Y), p1(X,V1), p2(Y,V2).`

then the set $P \cup \{\mathtt{G}\}$ of CHCs is satisfiable.

Algorithm $\mathcal{T}_{cata}$ introduces a new definition of the form:

`D: newf(V1,W1,V2,W2) :- f(X,Y), p1(X,V1), q1(X,W1), p2(Y,V2), q2(Y,W2).`

by using the catamorphisms $\mathtt{p}_1$ and $\mathtt{p}_2$ in `G` and adding (zero or more) new catamorphisms such as $\mathtt{q}_1$ and $\mathtt{q}_2$ coming from the contracts for other functions occurring in the program at hand. Let $P' \cup \{\mathtt{G}'\}$ be set of CHCs produced as output by algorithm $\mathcal{T}_{cata}$. Some of the CHCs in $\mathtt{P}'$ have `newf` as the head predicate and $\mathtt{G}'$ is the goal derived from goal `G`.

If $P' \cup \{\mathtt{G}'\}$ is proved to be satisfiable by any CHC solver, then there exists a model of $P' \cup \{\mathtt{G}'\}$ where the interpretation of $\mathtt{newf}(\mathtt{V1},\mathtt{W1},\mathtt{V2},\mathtt{W2})$ can be expressed as a constraint $\mathtt{d}(\mathtt{V1},\mathtt{W1},\mathtt{V2},\mathtt{W2})$, and hence

$$M(P') \models \forall\, \mathtt{V1},\mathtt{W1},\mathtt{V2},\mathtt{W2}.\ \mathtt{newf}(\mathtt{V1},\mathtt{W1},\mathtt{V2},\mathtt{W2}) \ \to\ \mathtt{d}(\mathtt{V1},\mathtt{W1},\mathtt{V2},\mathtt{W2})$$

where, for any given set S of definite clauses (i.e., clauses whose head is different from `false`), $M(S)$ denotes the least model of S. Now, by the soundness of the transformation [9],

$$M(P \cup \{\mathtt{D}\}) \models \forall\, \mathtt{V1},\mathtt{W1},\mathtt{V2},\mathtt{W2}.\ \mathtt{newf}(\mathtt{V1},\mathtt{W1},\mathtt{V2},\mathtt{W2}) \ \to\ \mathtt{d}(\mathtt{V1},\mathtt{W1},\mathtt{V2},\mathtt{W2})$$

and, by using clause `D`, we get:

$$M(P) \models \forall\, \mathtt{X},\mathtt{Y},\mathtt{V1},\mathtt{W1},\mathtt{V2},\mathtt{W2}.\ \mathtt{f}(\mathtt{X},\mathtt{Y}), \mathtt{p}_1(\mathtt{X},\mathtt{V1}), \mathtt{q}_1(\mathtt{X},\mathtt{W1}), \mathtt{p}_2(\mathtt{Y},\mathtt{V2}), \mathtt{q}_2(\mathtt{Y},\mathtt{W2}) \ \to\ \mathtt{d}(\mathtt{V1},\mathtt{W1},\mathtt{V2},\mathtt{W2})$$

By using also goal G, we get:

$$M(P) \models \forall\, \mathtt{X},\mathtt{Y},\mathtt{V1},\mathtt{W1},\mathtt{V2},\mathtt{W2}.\ \mathtt{f}(\mathtt{X},\mathtt{Y}),\ \mathtt{p}_1(\mathtt{X},\mathtt{V1}),\ \mathtt{q}_1(\mathtt{X},\mathtt{W1}),\ \mathtt{p}_2(\mathtt{Y},\mathtt{V2}),\ \mathtt{q}_2(\mathtt{Y},\mathtt{W2}) \ \rightarrow \\ \mathtt{c}(\mathtt{V1},\mathtt{V2})\ \&\ \mathtt{d}(\mathtt{V1},\mathtt{W1},\mathtt{V2},\mathtt{W2}) \qquad (\mathtt{SC})$$

which shows the validity of a strengthened contract for f. Indeed, since catamorphisms are total functions, the atoms $\mathtt{p}_1(\mathtt{X},\mathtt{V1})$, $\mathtt{q}_1(\mathtt{X},\mathtt{W1})$, $\mathtt{p}_2(\mathtt{Y},\mathtt{V2})$, and $\mathtt{q}_2(\mathtt{Y},\mathtt{W2})$ are satisfiable, for all values of X and Y, and the conclusion of SC is strengthened with respect to the conclusion of C. Notice that, however, even if the strengthened contract is valid, we have no guarantee that the verifier (e.g., STAINLESS) is able to prove it.

Finally, let us make a remark on the *minimality* of the strengthened contracts, related to the fact that the information provided by the models of the new predicates introduced by algorithm $\mathcal{T}_{cata}$ during program transformation, can also be used, so to say, in a partial way. Let us explain this point by referring to our program *Reverse*. In this case, instead of the strengthened postcondition for rev that is derived from the conjunction of the given postcondition is_dsorted(res) and formulas (3.1) and (3.2), we could have used the postcondition shown in Figure 7, which is derived from the given postcondition and formula (3.2) only.

```
... ensuring { res => is_dsorted(res) &&
               forall((n: BigInt) => (((hd(l)._2 >= n) ==> leq_all(n,res)))) }
```

Figure 7: Strengthened postcondition for rev. Version 2.

This postcondition is sufficiently strong to allow the STAINLESS verifier to prove the contracts for rev and snoc (actually, STAINLESS does take for that proof less time with respect to the time taken for the more complex postcondition of Figure 5). However, in general, the strengthened postconditions that do not take into account the whole information which is derivable from the model of new predicates, provide a less informative description of the behaviour of the program functions at hand. We leave for future work the issue of computing minimally strengthened postconditions.

5 Conclusions

A good software engineering practice requires us to associate a contract with every program function, that is, to write, for each program function, a precondition and a postcondition.

The current software technology provides, together with compilers (and interpreters), also program verifiers, so that given a program function and its contract, one can execute the function and also prove its partial or total correctness with respect to its contract. Due to undecidability results, there is no program verifier that is able to prove (or disprove) program correctness in all cases. However, often a verifier is not able to show correctness simply because the contracts have not been specified in an adequate manner.

We have addressed this problem and we have proposed a technique which is capable of improving the contracts, and in particular, it is capable of strengthening their postconditions, so that a given verifier is successful in proving the correctness of functions, while it is not successful when trying to prove their correctness using the original postconditons.

We have considered programs and contracts written in the functional fragment of Scala and we have considered the STAINLESS [14] verifier for Scala programs. Our technique is based on the translation of the program functions and their contracts for which STAINLESS is not successful, into a set of constrained Horn clauses (CHCs) [5, 13]. Then, in the case where programs manipulate Algebraic Data Structures, those clauses are transformed by VeriCaT [9], so that their satisfiability can hopefully be proved by

SPACER (or a different CHC solver) in the domain of integers and/or booleans. If satisfiability is proved and a model for those clauses is found (which is defined by constraints on integers and/or booleans), then via a final translation step, we derive from that model suitable strengthened postconditions for the contracts. These derived contracts are guaranteed to be valid and can hopefully be proved by the STAINLESS verifier. However, at the moment, we do not have any general result characterizing these successful cases.

The derivation of strengthened contracts is an important objective from a software engineering point of view, because strengthened contracts provide a uniform framework (the Scala framework, in our case) where to write programs and their contracts. In this uniform framework an automatic system, such as STAINLESS, can show that the given contracts are valid. This is an important feature, because contracts can be viewed both as a documentation of the programs and also as a specification of their behavior.

As mentioned above, the proof of satisfiability of the CHCs derived after translation from the given programs, is already a proof of the contracts, but that proof is not given in the same framework. Moreover, it should rely on the correctness of the translation from Scala programs to CHCs and also on the correctness of the transformation for ADT removal. In addition, the derivation of strengthened postconditions and their proof done by STAINLESS increase the reliability of the validity proofs of the contracts, in the sense that the satisfiability preserving transformation performed by VeriCaT and the model constructed by the CHC solver SPACER are shown to agree with the behaviour of the STAINLESS Scala verifier. This agreement of automatic tools is very important in practice, and in particular when we deal with large programs.

Currently, we are working towards the full mechanization of the two steps of our technique that are still performed manually, even though they are performed in a systematic way, namely: (i) the semantic preserving translation from Scala programs into constrained Horn clauses, and (ii) the construction of strengthened postconditons from the models of the satisfiable clauses provided by a CHC solver (such as SPACER or Eldarica).

Acknowledgments

The authors warmly thank the anonymous reviewers for their helpful comments and suggestions. The authors are members of the INdAM Research Group GNCS.

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 F. de Boer, M. M. Bonsangue, S. Graf & W.-P. de Roever, editors: *Formal Methods for Components and Objects*, Lecture Notes in Computer Science 4111, Springer, pp. 364–387, doi:10.1007/11804192_17.

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

[3] Grady Booch & Doug Bryan (1994): *Software engineering with Ada (3. ed.)*. Benjamin/Cummings series in object-oriented software engineering, Benjamin/Cummings.

[4] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma & Roberto Sebastiani (2013): *The MathSAT5 SMT solver*. In Nir Piterman & Scott Smolka, editors: *19th TACAS '13*, Lecture Notes in Computer Science 7795, Springer, pp. 93–107, doi:10.1007/978-3-642-36742-7_7.

[5] E. De Angelis, F. Fioravanti, J. P. Gallagher, M. V. Hermenegildo, A. Pettorossi & M. Proietti (2021): *Analysis and Transformation of Constrained Horn Clauses for Program Verification*. Theory and Practice of Logic Programming, pp. 1–69, doi:10.1017/S1471068421000211.

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

[7] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2020): *Removing Algebraic Data Types from Constrained Horn Clauses Using Difference Predicates*. In N. Peltier & V. Sofronie-Stokkermans, editors: Proceedings of the International Joint Conference on Automated Reasoning, IJCAR 2020, Lecture Notes in Artificial Intelligence 12166, Springer, pp. 83–102, doi:10.1007/978-3-030-51074-9_6.

[8] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2022a): *Satisfiability of constrained Horn clauses on algebraic data types: A transformation-based approach*. Journal of Logic and Computation 32, pp. 402–442, doi:10.1093/logcom/exab090.

[9] E. De Angelis, M. Proietti, F. Fioravanti & A. Pettorossi (2022): *Verifying Catamorphism-Based Contracts using Constrained Horn Clauses*. Theory and Practice of Logic Programming 22(4), pp. 555–572, doi:10.1017/S1471068422000175.

[10] J.-C. Filliâtre & A. Paskevich (2013): *Why3 - Where Programs Meet Provers*. In M. Felleisen & Ph. Gardner, editors: Programming Languages and Systems, 22nd European Symposium on Programming, ESOP'13, Rome, Italy, March 16–24, 2013, Lecture Notes in Computer Science 7792, Springer, pp. 125–128, doi:10.1007/978-3-642-37036-6_8.

[11] R. W. Floyd (1967): *Assigning Meanings to Programs*. In J. T. Schwartz, editor: Proceedings of Symposium on Applied Mathematics, Vol. 19, American Mathematical Society, Providence, R.I., USA, pp. 19–32, doi:10.1007/978-94-011-1793-7_4.

[12] H. Govind V. K., S. 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.

[13] S. Grebenshchikov, N. P. Lopes, C. Popeea & A. Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In: 33rd ACM SIGPLAN Conf. Programming Language Design and Implementation, PLDI '12, pp. 405–416, doi:10.1145/2345156.2254112.

[14] 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.

[15] 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*. Theory and Practice of Logic Programming 12(1–2), pp. 219–252, doi:10.1017/S1471068411000457.

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

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

[18] A. Komuravelli, A. Gurfinkel & S. Chaki (2014): *SMT-Based Model Checking for Recursive Programs*. In: 26th CAV '14, Lecture Notes in Computer Science 8559, Springer, pp. 17–34, doi:10.1007/978-3-319-08867-9_2.

[19] Yurii Kostyukov, Dmitry Mordvinov & Grigory Fedyukovich (2021): *Beyond the elementary representations of program invariants over algebraic data types*. In Stephen N. Freund & Eran Yahav, editors: PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021, ACM, pp. 451–465, doi:10.1145/3453483.3454055.

[20] 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.

[21] E. Meijer, M. M. Fokkinga & R. Paterson (1991): *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire*. In J. Hughes, editor: *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991*, Lecture Notes in Computer Science 523, Springer, pp. 124–144, doi:10.1007/3540543961_7.

[22] Bertrand Meyer (1991): *Eiffel: The Language*. Prentice-Hall.

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

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

[25] A. Reynolds & V. Kuncak (2015): *Induction for SMT Solvers*. In Deepak D'Souza, Akash Lal & Kim Guldstrand Larsen, editors: *16th VMCAI*, Lecture Notes in Computer Science 8931, Springer, pp. 80–98, doi:10.1007/978-3-662-46081-8_5.

[26] Solidity (2022): *Solidity v0.8.12 Documentation*. https://docs.soliditylang.org/.

[27] Philippe Suter, A. S. Köksal & V. Kuncak (2011): *Satisfiability Modulo Recursive Programs*. In E. Yahav, editor: *18th SAS '11*, Lecture Notes in Computer Science 6887, Springer, pp. 298–315, doi:10.1007/978-3-642-23702-7_23.

[28] H. Unno, S. Torii & H. Sakamoto (2017): *Automating Induction for Solving Horn Clauses*. In Rupak Majumdar & Viktor Kuncak, editors: *29th CAV '17, Part II*, Lecture Notes in Computer Science 10427, Springer, pp. 571–591, doi:10.1007/978-3-319-63390-9_30.

[29] Weikun Yang, Grigory Fedyukovich & Aarti Gupta (2019): *Lemma Synthesis for Automating Induction over Algebraic Data Types*. In Thomas Schiex & Simon de Givry, editors: *25th Int. Conf. Principles and Practice of Constraint Programming, CP 2019*, Lecture Notes in Computer Science 11802, Springer, pp. 600–617, doi:10.1007/978-3-030-30048-7_35.