Transformational Verification of Quicksort

Emanuele De Angelis
CNR-IASI
Via dei Taurini 19, 00185 Roma, Italy
emanuele.deangelis@iasi.cnr.it

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

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

Many transformation techniques developed for constraint logic programs, also known as *constrained Horn clauses* (CHCs), have found new useful applications in the field of program verification. In this paper, we work out a nontrivial case study through the transformation-based verification approach. We consider the familiar *Quicksort* program for sorting lists, written in a functional programming language, and we verify the pre/-postconditions that specify the intended correctness properties of the functions defined in the program. We verify these properties by: (1) translating them into CHCs, (2) transforming the CHCs by removing all list occurrences, and (3) checking the satisfiability of the transformed CHCs by using the Eldarica solver over booleans and integers. The transformation mentioned at Point (2) requires an extension of the algorithms for the elimination of inductively defined data structures presented in previous work, because during one stage of the transformation we use as lemmas some properties that have been proved at previous stages.

1 From Program Transformation to Program Verification

Program transformation gained a lot of popularity after the seminal paper by Burstall and Darlington [7], who advocated an approach based on *transformation rules*, which preserve the semantics of programs, and *transformation strategies*, which guide the application of the rules towards a goal of interest. This approach enables the separation, during program development, of the correctness issue from the efficiency issue.

Burstall and Darlington's rule-based approach has been proposed in the context of functional programming, and later extended to other programming paradigms, such as logic programming [35, 43] and constraint logic programming (CLP) [15]. The interest of applying program transformation techniques to declarative programming languages, like functional and logic programming, is due to the fact that in that context both specifications and programs are written as logical formulas, and program transformation can be viewed as a means for deriving, via logical deduction, efficient programs that are correct by construction [23].

Starting from the late 1990s, many program analysis and transformation techniques for logic and constraint logic programs have found new applications in the field of *program verification*. Initially, they have been applied to the proof of properties for abstract computational models such as *Petri nets*, *timed automata*, and *infinite state transition systems* [3, 14, 16, 18, 28, 39], and, later on, also for verifying programs written in concrete programming languages, including imperative and object-oriented languages [1, 9, 30, 34]. Indeed, logic programming, possibly extended with constraint theories, is a very suitable language for specifying program semantics and program properties [19, 34, 37]. Moreover, the notions of least and greatest models are the logical counterparts of least and greatest fixed points often used for program verification.

In the field of program verification, constraint logic programs are often called *constrained Horn clauses* (CHCs), when we want to stress their use as a reasoning formalism, rather than as a programming language [4]. The underlying constraint theories used in CHCs are typically those that axiomatize data structures used in programming, such as booleans, integer numbers, real numbers, bit vectors, arrays, heaps, and inductively defined data structures such as lists and trees. For checking the satisfiability of CHCs, effective *solvers*, such as *Eldarica* [24] and *Z3* [32] with the *Spacer* Horn engine [26], have been developed during the last years.

Several CHC transformations, including *fold/unfold* transformations and *CHC-specialisation*, have been applied to verification problems [9, 11, 12, 25, 31]. The basic idea is to transform a set of clauses P, whose satisfiability guarantees a certain program property, into a new set of clauses P', such that the satisfiability of P' (1) implies the satisfiability of P, and (2) is more effectively checked by the available CHC solver. One of these CHC transformations is the fold/unfold strategy for the elimination of inductively defined data structures from CHCs. This strategy was first proposed as a means for improving the efficiency of logic programs by avoiding intermediate data structures [38], and is strongly related to the well-known *deforestation* transformation in functional programming [45]. In the context of CHC verification, the advantage of eliminating inductively defined data structures is that the satisfiability of the derived clauses can be proved in simpler domains, such as the theory of booleans or the theory of linear arithmetic, for which existing solvers are very effective.

In previous work [12, 13], we have shown that, by eliminating inductively defined data structures from CHCs, we can avoid to extend solvers with induction-based inference rules, and yet we can prove universally quantified properties of programs acting on those structures. Indeed, experiments show that our two-step technique, consisting in preprocessing CHCs by eliminating inductively defined data structures, and then applying CHC solvers over booleans and integers, is competitive with respect to approaches based on extending solvers with induction over data structures [41, 44].

In this paper, we work out a case study through the transformation-based verification approach. We consider a program *Quicksort* for sorting lists, written in the pure functional fragment of *Scala* [33], implementing the familiar algorithm invented by Tony Hoare [21]. The program is equipped with *contracts*, i.e., pre/postconditions that specify the intended correctness properties of the various program functions. We check that the program verifies all contracts by: (1) translating them into CHCs, (2) transforming the CHCs by removing all list occurrences, and (3) checking the satisfiability of the transformed CHCs by using the Eldarica solver over booleans and integers. The transformation mentioned at Point (2) requires an extension of the algorithms for the elimination of inductively defined data structures presented in previous work, because during one stage of the transformation we will use as lemmas some contracts that we have verified at previous stages.

The advantage of our approach is that we avoid the use of very complex program verifiers, such as the STAINLESS tool developed for Scala [20], which combine reasoning in Hoare logic with induction and constraint solving, and instead, by our transformation, we reduce the verification task to a problem that can be handled by simpler CHC solvers. In fact, our specific *Quicksort* verification problem is not solved by STAINLESS.

The paper is organized as follows. In Section 2, we recall the transformation-based verification approach by considering the `partition` function, which is used by the *Quicksort* program. In Sections 3 and 4, we apply the transformation-based verification approach to the whole *Quicksort* program. In particular, in Section 3 we show how the problem of verifying the correctness of *Quicksort* with respect to its contracts is translated to CHCs. Then, in Section 4, we show how those CHCs are transformed by removing all list terms, hence deriving a set of clauses over booleans and integers whose satisfiability is proved by Eldarica. Finally, in Section 5, we compare our contribution to related work and we make some concluding considerations.

2 Program Verification via Constrained Horn Clause Transformation

In this section, we recall the transformation-based approach to program verification by means of a simple example. We consider a function `partition` for partitioning a list of natural numbers into two sublists by using a pivot element. This function will be used in the *Quicksort* program of Section 3. We translate the `partition` function into a set *PartitionCHCs* of clauses, and the contract associated with `partition` into a set *Gs* of *goals*, that is, clauses with `false` head. The satisfiability of $\mathit{PartitionCHCs} \cup \{G\}$, for all G in *Gs*, guarantees that `partition` is correct with respect to the specified contract. Then, for all G in *Gs*, we apply the transformation technique based on the Elimination Algorithm [12] for removing list occurrences from $\mathit{PartitionCHCs} \cup \{G\}$. The result of the transformation is a set T_G of clauses over the theories *Bool* of boolean values and *LIA* of linear integer arithmetic, which is satisfiable if and only if $\mathit{PartitionCHCs} \cup \{G\}$ is satisfiable. Finally, we check the satisfiability of T_G by using a CHC solver over *Bool* and *LIA*.

Let us consider the following program *Partition* written in the pure functional fragment of Scala [33]:

```
def all_grt(x: Nat, l: List[Nat]): Boolean = {
  l match {
    case Nil() => true
    case Cons(y, ys) if (x =< y) => false
    case Cons(y, ys) if (x > y) => all_grt(x, ys)
 }

def all_leq(x: Nat, l: List[Nat]): Boolean = {
  l match {
    case Nil() => true
    case Cons(y, ys) if (x > y) => false
    case Cons(y, ys) if (x =< y) => all_leq(x, ys)
  }
}

def partition(x: Nat, l: List[Nat]): (List[Nat], List[Nat]) = {
  l match {
    case Nil() => (Nil[Nat](), Nil[Nat]())
    case Cons(y, ys) =>
      val (l1, l2) = partition(x, ys)
      if (x > y) { (Cons(y, l1), l2) }
      else       { (l1, Cons(y, l2)) }
  }
} ensuring { res =>
  all_grt(x, res._1) && all_leq(x, res._2)  // partition postcondition
}
```

Listing 1: Program *Partition*. Variable `res` denotes the pair returned by the `partition` function, and `res._1` and `res._2` denote its first and second components, respectively.

Given a natural number `x` and a list `l` of natural numbers, we have that (i) `all_grt(x,l)` (and, respectively, `all_leq(x,l)`) returns `true` if `x` is greater than (respectively, less than or equal to) every element of `l`, and `false` otherwise, (ii) `partition(x,l)` returns a pair of lists `(l1,l2)`, where `l1` (respectively, `l2`) is the list of all the elements `y` of `l` such that `x` is greater than (respectively, less than or equal to) `y`. The `partition` function is annotated with a postcondition, specified by the `ensuring` assertion, which encodes the following contract:

$\forall$`x,l,l1,l2.  partition(x,l)==(l1,l2) ==> all_grt(x,l1) && all_leq(x,l2)` (Contract *Pivot*)

In general, a contract consists of a precondition, specified by a `require` assertion, and a postcondition, specified by an `ensuring` assertion. However, in the case of `partition`, the precondition is missing (i.e., it is `true`).

In order to prove that the contract *Pivot* is indeed satisfied, we first consider the translation of the *Partition* program into the following set *PartitionCHCs* of clauses (where natural numbers have been translated into non-negative integers in the *LIA* theory):

```
all_grt(X,[],B) :- X>=0, B=true.
all_grt(X,[Y|Ys],B) :- X=<Y, X>=0, B=false.
all_grt(X,[Y|Ys],B) :- X>Y, Y>=0, all_grt(X,Ys,B).

all_leq(X,[],B) :- X>=0, B=true.
all_leq(X,[Y|Ys],B) :- X>Y, Y>=0, B=false.
all_leq(X,[Y|Ys],B) :- X=<Y, X>=0, all_leq(X,Ys,B).

partition(X,[],[],[]).
partition(X,[Y|Ys],[Y|L1s],L2s) :- X>Y, Y>=0, partition(X,Ys,L1s,L2s).
partition(X,[Y|Ys],L1s,[Y|L2s]) :- X=<Y, X>=0, partition(X,Ys,L1s,L2s).
```

Listing 2: *PartitionCHCs*: Translation to CHCs of the *Partition* program.

The atoms (i) `all_grt(X,L,B)`, (ii) `all_leq(X,L,B)` and (iii) `partition(X,L,L1,L2)` hold in the least model of *PartitionCHCs* iff the expressions (i) `all_grt(X,L)==B`, (ii) `all_leq(X,L)==B` and (iii) `partition(X,L)==(L1,L2)`, respectively, hold in the functional program *Partition* of Listing 1. Contract *Pivot* is translated into the following two goals `G1` and `G2`, whose conjunction is equivalent to the contract:

```
false :- B=false, partition(X,L,L1,L2), all_grt(X,L1,B).                  % G1
false :- B=false, partition(X,L,L1,L2), all_leq(X,L2,B).                  % G2
```

Listing 3: CHC translation of the *Pivot* contract.

By a slight abuse of notation we use `false` to denote both the empty disjunction in the conclusion of a clause and a boolean value in a constraint. However, these two uses of `false` never generate any confusion. The use of the constraint `B=false` allows us to avoid negative literals in the body of goals, and hence to stick to Horn format. The satisfiability of *PartitionCHCs* $\cup$ {G}, for all $G \in$ {`G1`,`G2`}, guarantees that `partition` satisfies the contract *Pivot*.

Let us consider *PartitionCHCs* $\cup$ {`G2`} (the satisfiability of *PartitionCHCs* $\cup$ {`G1`} can be proved in a similar way). Unfortunately, *PartitionCHCs* $\cup$ {`G2`} cannot be proved satisfiable by state-of-the-art CHC solvers, such as Eldarica or Z3, because they do not use any method, such as induction on the list structure, which would be needed for reasoning on universally quantified list properties (goals, and in general clauses, have an implicit universal quantification in front).

To overcome this difficulty, we now apply the Elimination Algorithm, which uses the *definition*, *unfolding*, and *folding* rules [15, 43], and from *PartitionCHCs* $\cup$ {`G2`} we derive an equisatisfiable set

$T_{\mathtt{G2}}$ of CHCs where lists do not occur. In this way, we can check the satisfiability of the transformed CHCs $T_{\mathtt{G2}}$ using a solver over *Bool* and *LIA*, without the need for any induction-based method for reasoning on lists. We start off by introducing a new predicate `pl` defined by the following clause (the variable names are automatically generated by the interactive transformation system MAP [40]):

```
1.  pl(A,B) :- partition(B,C,D,E), all_leq(B,E,A).
```

and we eliminate list terms from goal `G2` by folding it using clause 1 as follows:

```
F.  false :- B=false, pl(X,B).
```

Now, we look for a recursive definition of predicate `pl` without occurrences of lists. By unfolding clause 1 with respect to the `partition` atom, we obtain

```
2.  pl(A,B) :- all_leq(B,[],A).
3.  pl(A,B) :- B>=0, partition(B,D,E,F), all_leq(B,F,A).
4.  pl(A,B) :- B=<C, B>=0 partition(B,D,E,F), all_leq(B,[C|F],A).
```

We proceed by unfolding clauses 2 and 4 with respect to `all_leq` atoms, thereby obtaining

```
5.  pl(A,B) :- A=true, B>=0.
6.  pl(A,B) :- B=<C, B>=0, partition(B,D,E,F), B>C, A=false.
7.  pl(A,B) :- B=<C, B>=0, partition(B,D,E,F), B=<C, all_leq(B,F,A).
```

We remove clause 6 because it contains an unsatisfiable constraint. Moreover, clause 7 is equal to clause 3, modulo equivalence of constraints, and thus we remove it. As a final step, we use the definition clause 1 for folding clause 3, hence deriving the following final set $T_{\mathtt{G2}}$ of CHCs:

```
5.  pl(A,B) :- A=true, B>=0.
8.  pl(A,B) :- B>=0, pl(A,B).
F.  false :- A=false, pl(A,B).
```

The set $T_{\mathtt{G2}}$ is satisfiable, and Eldarica easily finds that `pl(A,B) :- A=true, B>=0` is a *model* for $T_{\mathtt{G2}}$. Indeed, by replacing each occurrence of `pl(A,B)` by `(A=true, B>=0)` in the clauses of $T_{\mathtt{G2}}$, we derive clauses that are true in the combined theory of booleans and integers.

3 Specification of Quicksort with Parameterized Catamorphisms

Now, we consider the following program that implements the *Quicksort* algorithm:

```
def quicksort(l: List[Nat]): List[Nat] = {
  l match {
    case Nil() => Nil[Nat]()
    case Cons(x, xs) =>
      val (ys,zs) = partition(x, xs)
      append(quicksort(ys), Cons(x, quicksort(zs)))
  }
} ensuring { res =>
    forall((a: Nat) => all_grt(a,l) ==> all_grt(a,res)) &&
    forall((a: Nat) => all_leq(a,l) ==> all_leq(a,res)) &&
    isSorted(0,res) &&
    forall((a: Nat) => count(a,l) == count(a,res))
}

def append(l: List[Nat], ys: List[Nat]): List[Nat] = {
  require( isSorted(0,l) && ( ys == Nil()
    ( all_grt(ys.head,l) && all_leq(ys.head,ys.tail) && isSorted(0,ys.tail) )))
  l match {
    case Nil() => ys
    case Cons(x, xs) => Cons(x, append(xs,ys))
  }
} ensuring { res => isSorted(0,res) }
```

Listing 4: Program *Quicksort*.

In program *Quicksort*, the function `partition` is defined as in Listing 1. The variable `res` denotes the return value of a given function. The `require` and `ensuring` assertions specify the preconditions and postconditions of the contracts for the `quicksort` and `append` functions. The contract specifications use the functions `all_grt` and `all_leq` defined in Listing 1, and also the functions `count` and `isSorted` defined below.

```
def count(a: Nat, l: List[Nat]): Nat = {
  l match {
    case Nil() => 0
    case Cons(x, xs) => if (x==a) { count(a,xs)+1 } else { count(a,xs) }
  }
}

def isSorted(a: Nat, l: List[Nat]): Boolean = {
  l match {
    case Nil() => true
    case Cons(x,xs) => if (a<=x) isSorted(x,xs) else false
  }
}
```

Listing 5: Auxiliary functions for the *Quicksort* contracts.

All of the functions used in the contract specifications have a common recursive pattern, which slightly extends the *catamorphism* pattern defined in functional programming [29]. Indeed, the functions considered here admit an extra parameter, and are called *parameterized catamorphisms*. In particular, the function `isSorted` is defined by induction on the list structure by considering the two cases where the input list `l` is either `Nil()` or `Cons(x,xs)`. By using the extra parameter `a` we avoid to split the case `Cons(x,xs)` into `Cons(x,Nil())` and `Cons(x,Cons(y,xs))`, and we express the sortedness of list `l` as `isSorted(0,l)` (recall that the elements of `l` are all nonnegative numbers). The general pattern of parameterized catamorphisms is defined below.

```
def pCata(p:A, l:List[A]): B = {
  match l {
    case Nil() => c
    case Cons(x,xs) => g(p,x,pCata(h(p,x),xs))
  }
}
```

Listing 6: General form of parameterized catamorphism on `List[A]`.

In Listing 6, (i) `A` is any type and `B` is the type of the integer or boolean values, (ii) `c` is a constant of type `B`, (iii) `g` is a total, `B`-valued function, and (iv) `h` is a total, `A`-valued function. Thus, also `pCata` is a total, `B`-valued function.

The task of verifying the contract for a function `f` consists in proving the validity of a universally quantified implication of the form:

```
  ∀x̄. pre(x̄) ==> post(x̄,f(x̄))
```

where: (i) `x̄` is a tuple of variables (a subset of the function inputs), and (ii) `pre(x̄)` and `post(x̄,f(x̄))` are the precondition and postcondition, respectively, specified by the `require` and `ensuring` assertions using parameterized catamorphisms.

The `pre(x̄)` assertion for `quicksort` is absent. Thus, verifying the contract of `quicksort` consists in verifying the validity of `∀l. true ==> post(l,quicksort(l))`, where `post(l,quicksort(l))` is the conjunction of the following assertions:

```
1. ∀a. all_grt(a,l) ==> all_grt(a,quicksort(l)))
2. ∀a. all_leq(a,l) ==> all_leq(a,quicksort(l)))
3. isSorted(0,quicksort(l))
4. ∀a. count(a,l) == count(a,quicksort(l)))
```

Assertions 1 and 2 state that `quicksort` preserves the postcondition of the function `partition`. Assertion 3 expresses the sortedness property. Assertion 4 states that the multiset of natural numbers in the input list `l` is the same as the multiset of the elements in `quicksort(l)`.

For the function append, the precondition `pre(l,ys)`, where `l` and `ys` are the input lists, is defined as follows:

```
isSorted(0,l) && ( ys == Nil() ||
  ( all_grt(ys.head,l) && all_leq(ys.head,ys.tail) && isSorted(0,ys.tail) ) )
```

The assertion states that (i) `l` is sorted, and either (ii) `ys` is the empty list or (iii.1) the head of `ys` is greater than every element of `l`, (iii.2) the head of `ys` is less than or equal to every element occurring in its tail, and (iii.3) the tail of `ys` is sorted. The postcondition of the function append states that its output is a sorted list.

The *Quicksort* program (Listing 4) and the auxiliary functions (Listing 5) are translated to the set *QuicksortCHCs* of clauses in Listing 7 below.

```
quicksort([],[]).
quicksort([X|Xs],Ys) :- X>=0,
  partition(X,Xs,Littles,Bigs),
  quicksort(Littles,Ls), quicksort(Bigs,Bs),
  append(Ls,[X|Bs],Ys).

append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) :- X>=0, append(Xs,Ys,Zs).

count(X,[],N) :- X>=0, N=0.
count(X,[Y|Ys],N) :- X>=0, X=Y, N=M+1, count(X,Ys,M).
count(X,[Y|Ys],N) :- X>=0, Y>=0, X=\=Y, N=M, count(X,Ys,M).

isSorted(A,[],B) :- A>=0, B=true.
isSorted(A,[X|Xs],B) :- X>=0, A>X, B=false.
isSorted(A,[X|Xs],B) :- A>=0, A=<X, isSorted(X,Xs,B).
```

Listing 7: *QuicksortCHCs*: CHC translation of *Quicksort* and its auxiliary functions.

The contracts are translated to CHC goals as follows.

```
% quicksort contract
false :- B1=true, B2=false, all_grt(A,B,B1), quicksort(B,C), all_grt(A,C,B2).   % G3
false :- B1=true, B2=false, all_leq(A,B,B1), quicksort(B,C), all_leq(A,C,B2).   % G4
false :- B1=false, quicksort(L,S), isSorted(0,S,B1).                            % G5
false :- N1=/=N2, count(X,L,N1), quicksort(L,S), count(X,S,N2).                 % G6
```

```
% append contract
false :- B1=true, B2=true, B3=true, B4=true, B5=false,                    % G7
  all_grt(X,Xs,B1), isSorted(0,Xs,B2),
  all_leq(X,Ys,B3), isSorted(0,Ys,B4),
  append(Xs,[X|Ys],Zs), isSorted(0,Zs,B5).
```

Listing 8: CHC translation of the *Quicksort* contracts.

Similarly to the translation of the contract for the *Partition* program, the use of boolean constraints avoids the introduction of negative literals.

Now, to prove the correctness of *Quicksort* with respect to its contracts, it remains to show that *QuicksortCHCs* $\cup \{G\}$ is satisfiable for all goals $G \in \{$`G3,G4,G5,G6,G7`$\}$. Unfortunately, these satisfiability problems cannot be directly solved by Eldarica or Z3.

4 Removing List Arguments

Similarly to the `partition` example presented in Section 2, the proof of satisfiability of the set of clauses *QuicksortCHCs* $\cup \{G\}$, where G is a goal among `G3,G4,G5,G6,G7`, proceeds in two steps. First, we transform *QuicksortCHCs* $\cup \{G\}$ by using the fold/unfold rules, and derive a new set T_G such that: (i) T_G is a set of CHCs over *LIA* and *Bool*, without any list argument, and (ii) if T_G is satisfiable, then *QuicksortCHCs* $\cup \{G\}$ is satisfiable. Then, we check the satisfiability of T_G by using a CHC solver over *LIA* and *Bool*.

The main difference with respect to the `partition` example is that we also use as lemmas the properties that we have already proved in previous applications of our method. For instance, having proved that *PartitionCHCs* $\cup \{$`G2`$\}$ is satisfiable (see Section 2), during subsequent transformations we can use the property

```
∀X,L,L1,L2. partition(X,L,L1,L2) ==> all_leq(X,L2,true)
```

and add (instances of) `all_leq(X,L2,true)` to the body of a clause where `partition(X,L,L1,L2)` occurs.

The general form of the transformation strategy that we apply to eliminate list terms is an extension of the Elimination Algorithm [12]. The strategy is parametric with respect to specific *Define-Fold*, *Unfold*, and *Replace*$_{cata}$ functions.

List Removal $\mathscr{R}_{cata}$.
Input: A set *Cls* $\cup \{G\}$, where *Cls* is a set of non-goal clauses and G is a goal, and a set *Props* of properties in the form of implications `B1 ==> B2`;
Output: A set T_G of clauses over *LIA* and *Bool* such that if T_G is satisfiable, then *Cls* $\cup \{G\}$ is satisfiable.

```
$Defs := \emptyset$;   $InCls := \{G\}$;   $T_G := \emptyset$;
while $InCls \neq \emptyset$ do
    $(NewDefs, FldCls) := Define\text{-}Fold(Defs, InCls)$;
    $UnfCls := Unfold(NewDefs, Cls)$;
    $RCls := Replace_{cata}(UnfCls, Props)$;
    $Defs := Defs \cup NewDefs$;   $InCls := RCls$;   $T_G := T_G \cup FldCls$;
```

In $\mathscr{R}_{cata}$, the set *Defs* of clauses stores the new definitions introduced during the application of the transformation strategy. The set *InCls* is the set of clauses to be transformed. T_G is the set of transformed clauses. *NewDefs* and *FldCls* are the sets of clauses derived by applying the definition and folding rules, respectively using the function *Define-Fold*. *UnfCls* is the set of clauses derived by applying the unfolding rule using the function *Unfold*. *RCls* is the set of clauses derived by applying the function $Replace_{cata}$, which uses properties stored in *Props* corresponding to goals whose satisfiability has been proved in previous steps.

Let us explain the list removal strategy in action for the transformation of *QuicksortCHCs* $\cup$ {G5}, where we also use the properties *Props* corresponding to goals G1,G2,G3,G4. The properties corresponding to goals G6 and G7 are not needed for G5.

Define-Fold. $\mathscr{R}_{cata}$ starts off by introducing the following new predicate:

```
1.  qss(A) :- quicksort(B,C), isSorted(0,C,A).
```

and folding the goal G5 as follows:

```
F5.  false :- A=false, qss(A).
```

Unfold. By unfolding clause 1 with respect to the `quicksort` and the `isSorted` atoms, we get:

```
2. qss(true).
3. qss(A) :- B>=0,
       partition(B,C,D,E), quicksort(D,F), quicksort(E,G),
       append(F,[B|G],H), isSorted(0,H,A).
```

$Replace_{cata}$. Now, we apply the properties corresponding to goals G1 and G2, which translate the post-condition of the `partition` function (see Section 2), and we add the two atoms `all_grt(B,D,true)` and `all_leq(B,E,true)` to the body of clause 3:

```
4. qss(A) :- B>=0,
       partition(B,C,D,E), all_grt(B,D,true), all_leq(B,E,true),
       quicksort(D,F), quicksort(E,G), append(F,[B|G],H), isSorted(0,H,A).
```

Next, by using G3 and G4, we add the atoms `all_grt(B,F,true)` and `all_leq(B,G,true)` to the body of clause 4 and we derive:

```
5. qss(A) :- B>=0,
       partition(B,C,D,E), all_grt(B,D,true), all_leq(B,E,true),
       quicksort(D,F), all_grt(B,F,true),
       quicksort(E,G), all_leq(B,G,true),
       append(F,[B|G],H), isSorted(0,H,A).
```

Now, in order to fold the two `quicksort` atoms using clause 1, we add two instances of the parameterized catamorphism `isSorted`, where the output boolean value is an unbound variable, and hence implicitly existentially quantified. This step is correct because, by the totality of the `isSorted` function, the following property holds: $\forall$`L:List[Nat]` $\exists$`B:Boolean. isSorted(0,L,B)`.

Hence, we get:

```
6. qss(A) :- B>=0,
       partition(B,C,D,E), all_grt(B,D,true), all_leq(B,E,true),
       quicksort(D,F), isSorted(0,F,B1), all_grt(B,F,true),
       quicksort(E,G), isSorted(0,G,B2), all_leq(B,G,true),
       append(F,[B|G],H), isSorted(0,H,A).
```

Note that we cannot use the property corresponding to goal G7 because B1 and B2 are unbound variables, while G7 requires them to be bound to `true`.

Now, we perform a second iteration of the List Removal strategy.

Define-Fold. We fold twice clause 6 using clause 1, and we get:

```
7. qss(A) :- B>=0,
       partition(B,C,D,E), all_grt(B,D,true), all_leq(B,E,true),
       qss(B1), isSorted(0,F,B1), all_grt(B,F,true),
       qss(B2), isSorted(0,G,B2), all_leq(B,G,true),
       append(F,[B|G],H), isSorted(0,H,A).
```

By this folding step, we do not remove the `isSorted` atoms, which share the lists `F` and `G` with the `append` atom. In contrast, we remove the conjunction `partition(B,C,D,E),all_grt(B,D,true), all_leq(B,E,true)`, which, by the totality of `partition(B,C,D,E)` and by the properties corresponding to goals `G1` and `G2`, is always true:

```
8. qss(A) :- B>=0,
       qss(B1), isSorted(0,F,B1), all_grt(B,F,true),
       qss(B2), isSorted(0,G,B2), all_leq(B,G,true),
       append(F,[B|G],H), isSorted(0,H,A).
```

Then, we introduce the following new definition:

```
9. a(B,X,Y,Z,T,U,B1,B2,A) :-
       isSorted(X,F,B1), all_grt(B,F,T),
       isSorted(Y,G,B2), all_leq(B,G,U),
       append(F,[B|G],H), isSorted(Z,H,A).
```

which we use for folding clause 8, hence deriving:

```
10. qss(A) :- B>=0, qss(B1), qss(B2), a(B,0,0,0,true,true,B1,B2,A).
```

Now, predicate `qss` is defined by clauses 2 and 10, which have no lists. However, predicate `a`, occurring in the body of clause 10, is defined by clause 9, whose body has some occurrences of list terms. Thus, the List Removal strategy continues by transforming clause 9 and, after a few iterations, produces a set of clauses without lists. The final result of this transformation is a set $T_{\mathtt{G5}}$ including goal `F5`, clauses 2 and 10, and the clauses for predicate `a` (and some extra predicates introduced in subsequent iterations) reported in the Appendix. $T_{\mathtt{G5}}$ is a set of Horn clauses with constraints in *LIA* and *Bool* only.

The CHC solver Eldarica is able to prove the satisfiability of $T_{\mathtt{G5}}$, and hence also the initial set of clauses *QuicksortCHCs* $\cup$ {`G5`} is satisfiable. Similarly, by applying again the List Removal strategy and then proving satisfiability by Eldarica over *LIA* and *Bool*, we are able to verify all contracts of the *Quicksort* program.

We have also attempted to verify the same contracts by using the STAINLESS system [20], a verifier for the Scala language. STAINLESS is able to verify the contracts of the functions `partition` and `append`, but not the one of `quicksort`.

5 Related Work and Conclusions

The *Quicksort* algorithm is a brilliant invention by Tony Hoare, presented in his famous 1961 paper [21]. A formal proof of partial correctness, using the axiomatic approach [22], was presented by Hoare himself, in a joint paper with M. Foley [17]. Since then, many hand-made proofs have been worked out, for several variants (both recursive and iterative) of the algorithm (see, for instance, the book by Apt et al. [2]). Also semi-automated proofs have been presented, using *program verifiers* that implement Hoare logic, such as DAFNY [8, 27] and STAINLESS [20]. However, the success of program verifiers is very much dependent on the assertions provided by the programmer. In particular, we have checked that STAINLESS is able to verify the contracts of a program implementing a variant of *Quicksort* [1], but it could not verify the version presented in Section 3 of this paper.

Also our proof depends critically on the contract specifications, because we first prove and then use them as lemmas during the transformation phase. For instance, a crucial role is played by the postcondition of the `partition` function, that is, contract *Pivot* of Section 2:

```
∀x,l,l1,l2.  partition(x,l)==(l1,l2) ==> all_grt(x,l1) && all_leq(x,l2)
```

stating that the output of `partition` is a pair of lists `(l1,l2)` such that the pivot `x` is greater than all elements in `l1`, and smaller or equal than all elements in `l2`. Without introducing the two predicates `all_grt` and `all_leq`, and then proving that they are preserved by applications of the `quicksort` function, our transformation would not go through.

Another interesting point is that in all contract specifications we use predicates defined by a simple induction scheme, which we have called parameterized catamorphisms. This form helps introducing suitable new predicates (the famous *eureka definitions* in Burstall and Darlington's approach [7]). Indeed, all predicates introduced by the definition rule in our transformations (including the ones not shown in the paper) are defined as a conjunction of an atom, representing a call to a program function, and one or more atoms representing parameterized catamorphism. We argue that, by exploiting properties of parameterized catamorphisms, one can develop a fully automatic version of the transformation strategy $\mathscr{R}_{cata}$ that always succeeds in eliminating lists and, more in general, inductively defined data structures, from large classes of CHCs. We leave this task for future research.

Catamorphisms (on trees) were used in the context of *Satisfiability Modulo Theories*, to define satisfiability algorithms that terminate for suitable classes of formulas [36, 42]. A special form of integer-valued catamorphisms, such as *list length*, *term-size*, and in general, the so-called *type-based norms*, are used by techniques for proving termination of logic programs [5]. Our definition of parameterized catamorphism slightly extends the one of list catamorphism usually given in the context of functional programming [29]. Our definition allows an extra parameter, which makes the inductive scheme a little more flexible.

A more challenging problem is to discover pre/postconditions defined by catamorphisms which are not provided by the programmer. For instance, suppose that for the *Quicksort* program the programmer only specifies the contract for the main function `quicksort` using the functions `isSorted` and `count`. Then, an automated verifier (or transformer) should be able to discover suitable pre/postconditions such as the ones we have provided in terms of predicates `all_grt` and `all_leq`. This problem is related to the discovery of suitable lemmata during automated theorem proving [6] and program transformation [13], which is well-known to be very hard. However, we argue that, restricting the search for those lemmata among (parameterized) catamorphisms of suitable form, could be a fruitful heuristic.

Acknowledgment

We shared most of our scientific careers, starting from their beginnings, with Alberto Pettorossi, and the influence of his approach to Science on our own way of doing research has been enormous. First of all, the topics he contributed to explore starting from the 1970s, such as program transformation, program verification and, in general, the use of logic and formal methods in computing, are still central in our current research, as witnessed by the present paper. But, much more than that, Alberto's everyday example taught us the commitment and honesty in doing research, looking at the substance and not at the

[1] See the verification benchmarks at `https://github.com/epfl-lara/stainless/`.

surface, without, however, neglecting form and beauty. Indeed, Alberto's classical studies at secondary school, Latin and Greek ancient languages, as well as Italian literature classics, had a big impact on his way of writing papers. "We must love our readers" is one of his recurrent sentences! Finally, we want to say that Alberto's teachings go far beyond the scientific side: through his continuous emotional support of young and weak people, he has always shown the joy of committing one's life to something valuable.

Thanks Alberto, our teacher, colleague and friend.

We would also like to thank Laurent Fribourg, Alexei Lisitsa, and Andrei Nemytykh, for organizing this workshop dedicated to Alberto and inviting us to write this paper. This work has been partially supported by INdAM-GNCS.

References

[1] E. Albert, M. Gómez-Zamalloa, L. Hubert & G. Puebla (2007): *Verification of Java Bytecode Using Analysis and Transformation of Logic Programs*. In M. Hanus, editor: *Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science 4354, Springer, pp. 124–139, doi:10.1007/978-3-540-69611-7_8.

[2] K. R. Apt, F. S. de Boer & E.-R. Olderog (2009): *Verification of Sequential and Concurrent Programs*, Third edition. Springer, doi:10.1007/978-1-84882-745-5.

[3] G. Banda & J. P. Gallagher (2009): *Analysis of Linear Hybrid Systems in CLP*. In Michael Hanus, editor: *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Valencia, Spain, July 17–18, 2008, Revised Selected Papers*, Lecture Notes in Computer Science 5438, Springer, pp. 55–70, doi:10.1007/978-3-642-00515-2_5.

[4] N. Bjørner, A. Gurfinkel, K. L. McMillan & A. Rybalchenko (2015): *Horn Clause Solvers for Program Verification*. In L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner & W. Schulte, editors: *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, Lecture Notes in Computer Science 9300, Springer, pp. 24–51, doi:10.1007/978-3-319-23534-9_2.

[5] M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim & W. Vanhoof (2007): *Termination analysis of logic programs through combination of type-based norms*. *ACM Trans. Program. Lang. Syst.* 29(2), pp. 10–es, doi:10.1145/1216374.1216378.

[6] 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.

[7] R. M. Burstall & J. Darlington (1977): *A Transformation System for Developing Recursive Programs*. *Journal of the ACM* 24(1), pp. 44–67, doi:10.1145/321992.321996.

[8] R. Certezeanu, S. Drossopoulou, B. Egelund-Müller, K. R. M. Leino, S. Sivarajan & M. J. Wheelhouse (2016): *Quicksort Revisited - Verifying Alternative Versions of Quicksort*. In E. Ábrahám, M. M. Bonsangue & E. Broch Johnsen, editors: *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science 9660, Springer, pp. 407–426, doi:10.1007/978-3-319-30734-3_27.

[9] 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.

[10] 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, doi:10.1007/978-3-642-54862-8_47. Available at http://www.map.uniroma2.it/VeriMAP.

[11] 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`.

[12] 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`. Special Issue on *ICLP '18*.

[13] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2020): *Removing Algebraic Data Types from Constrained Horn Clauses Using Difference Predicates*. In: *Proceedings of the International Joint Conference on Automated Reasoning, IJCAR '20*, Lecture Notes in Computer Science 12166, Springer, Cham, pp. 83–102, doi:`10.1007/978-3-030-51074-9_6`.

[14] G. Delzanno & A. Podelski (1999): *Model Checking in CLP*. In R. Cleaveland, editor: *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '99*, Lecture Notes in Computer Science 1579, Springer-Verlag, pp. 223–239, doi:`10.1007/3-540-49059-0_16`.

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

[16] F. Fioravanti, A. Pettorossi & M. Proietti (2001): *Verifying CTL Properties of Infinite State Systems by Specializing Constraint Logic Programs*. In: *Proceedings of the ACM SIGPLAN Workshop on Verification and Computational Logic VCL'01, Florence (Italy)*, Technical Report DSSE-TR-2001-3, University of Southampton, UK, pp. 85–96.

[17] M. Foley & C. A. R. Hoare (1971): *Proof of a Recursive Program: Quicksort*. Comput. J. 14(4), pp. 391–395, doi:`10.1093/comjnl/14.4.391`.

[18] L. Fribourg & H. Olsén (1997): *A decompositional approach for computing least fixed-points of Datalog programs with Z-counters*. Constraints 2(3/4), pp. 305–335, doi:`10.1023/A:1009747629591`.

[19] 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`.

[20] 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`.

[21] C. A. R. Hoare (1961): *Algorithm 64: Quicksort*. Commun. ACM 4(7), p. 321, doi:`10.1145/366622.366644`.

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

[23] C. J. Hogger (1981): *Derivation of Logic Programs*. Journal of the ACM 28(2), pp. 372–392, doi:`10.1145/322248.322258`.

[24] H. Hojjat & Ph. Rümmer (2018): *The ELDARICA Horn Solver*. In N. Bjørner & A. Gurfinkel, editors: *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 – November 2, 2018*, IEEE, pp. 1–7, doi:`10.23919/FMCAD.2018.8603013`.

[25] B. Kafle & J. P. Gallagher (2017): *Constraint specialisation in Horn clause verification*. Sci. Comput. Program. 137, pp. 125–140, doi:`10.1016/j.scico.2017.01.002`.

[26] A. Komuravelli, A. Gurfinkel, S. Chaki & E. M. Clarke (2013): *Automatic Abstraction in SMT-Based Unbounded Software Model Checking*. In N. Sharygina & H. Veith, editors: *Computer Aided Verification, Proceedings of the 25th International Conference CAV '13, Saint Petersburg, Russia, July 13–19, 2013*, Lecture Notes in Computer Science 8044, Springer, pp. 846–862, doi:`10.1007/978-3-642-39799-8_59`.

[27] K. R. M. Leino (2013): *Developing Verified Programs with Dafny*. In: *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, IEEE Press, pp. 1488–1490, doi:`10.1109/ICSE.2013.6606754`.

[28] M. Leuschel & H. Lehmann (2000): *Coverability of Reset Petri Nets and Other Well-Structured Transition Systems by Partial Deduction.* 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. 101–115, doi:10.1007/3-540-44957-4_7.

[29] 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, Proceedings*, Lecture Notes in Computer Science 523, Springer, pp. 124–144, doi:10.1007/3540543961_7.

[30] M. Méndez-Lojo, J. A. Navas & M. V. Hermenegildo (2008): *A Flexible, (C)LP-Based Approach to the Analysis of Object-Oriented Programs.* In: *17th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR '07, Kongens Lyngby, Denmark, August 23–24, 2007*, Lecture Notes in Computer Science 4915, Springer, pp. 154–168, doi:10.1007/978-3-540-78769-3_11.

[31] 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. Available at http://www.easychair.org/publications/paper/340359.

[32] 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, doi:10.1007/978-3-540-78800-3_24.

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

[34] J. C. Peralta, J. P. Gallagher & H. Saglam (1998): *Analysis of Imperative Programs through Analysis of Constraint Logic Programs.* In G. Levi, editor: *Proceedings of the 5th International Symposium on Static Analysis, SAS '98*, Lecture Notes in Computer Science 1503, Springer, pp. 246–261, doi:10.1007/3-540-49727-7_15.

[35] A. Pettorossi & M. Proietti (1989): *Decidability Results and Characterization of Strategies for the Development of Logic Programs.* In G. Levi & M. Martelli, editors: *Proceedings of the Sixth International Conference on Logic Programming, Lisbon, Portugal*, The MIT Press, pp. 539–553.

[36] Tuan-Hung Pham, Andrew Gacek & Michael 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.

[37] A. Podelski & A. Rybalchenko (2007): *ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement.* In M. Hanus, editor: *Practical Aspects of Declarative Languages, PADL '07*, Lecture Notes in Computer Science 4354, Springer, pp. 245–259, doi:10.1007/978-3-540-69611-7_16.

[38] M. Proietti & A. Pettorossi (1995): *Unfolding-Definition-Folding, in this Order, for Avoiding Unnecessary Variables in Logic Programs.* *Theoretical Computer Science* 142(1), pp. 89–124, doi:10.1016/0304-3975(94)00227-A.

[39] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift & D. S. Warren (1997): *Efficient Model Checking Using Tabled Resolution.* In: *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, Lecture Notes in Computer Science 1254, Springer-Verlag, pp. 143–154, doi:10.1007/3-540-63166-6_16.

[40] S. Renault, A. Pettorossi & M. Proietti (1998): *Design, Implementation, and Use of the MAP Transformation System.* R 491, IASI-CNR, Rome, Italy. Available at http://www.iasi.cnr.it/~proietti/system.html.

[41] 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, January 12–14, 2015*, Lecture Notes in Computer Science 8931, Springer, pp. 80–98, doi:10.1007/978-3-662-46081-8_5.

[42] Ph. Suter, M. Dotta & V. Kuncak (2010): *Decision procedures for algebraic data types with abstractions*. In M. V. Hermenegildo & J. Palsberg, editors: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, ACM, pp. 199–210, doi:10.1145/1706299.1706325.

[43] 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.

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

[45] P. L. Wadler (1990): *Deforestation: Transforming Programs to Eliminate Trees*. *Theoretical Computer Science* 73, pp. 231–248, doi:10.1016/0304-3975(90)90147-A.

Appendix

We list below the final set $T_{\texttt{G5}}$ of clauses derived from *QuicksortCHCs* ∪ {G5}. Clauses 11–28 have been derived automatically from clause 9, by using the implementation of the Elimination Algorithm on the VeriMAP [2] system [10]. Both Eldarica and Z3 are able to prove the satisfiability of this set of clauses.

```
F5. false :- A=false, qss(A).
2.  qss(A) :- A=true.
10. qss(A) :- B>=0, qss(B1), qss(B2), a(B,0,0,0,true,true,B1,B2,A).
11. a(A,B,C,D,E,F,G,H,I) :- A=J, B=0, C=0, D=0, E=true, F=true, G=K, H=L, I=M, N=0,
                 O=0, P=0, Q=J, R=true, S=J, T=true, J>=0, new2(J,T,Q,S,R,P,K,O,L,N,M).
12. new2(A,B,C,D,E,F,G,H,I,J,K) :- A=L, B=true, C=L, D=L, E=true, F=0, G=true, H=0,
                 J=0, K=M, L>=0, new3(L,M,D,E,H,I).
13. new2(A,B,C,D,E,F,G,H,I,J,K) :- A=L, B=true, C=L, D=L, E=true, F=0, G=M, H=0,
                 J=0, K=N, O=true, P=Q, Q-L=< -1, Q>=0, new6(C,L,O,P,M,Q,N,D,E,H,I).
14. new3(A,B,C,D,E,F) :- A=C, B=true, D=true, E=0, F=true, C>=0.
15. new3(A,B,C,D,E,F) :- A=G, B=H, C=G, D=true, E=0, F=H, I=true, G>=0, J-G>=0,
                 new10(G,I,J,H).
16. new6(A,B,C,D,E,F,G,H,I,J,K) :- A=L, B=L, C=true, D=F, E=true, G=M, H=L, I=true,
                 J=0, F>=0, L-F>=0, new7(L,M,H,I,J,K).
17. new6(A,B,C,D,E,F,G,H,I,J,K) :- A=L, B=L, C=true, D=F, E=false, G=false, H=L,
                 I=true, J=0, M=true, F>=1, L-F>=0, new9(A,L,M,H,I,J,K).
18. new6(A,B,C,D,E,F,G,H,I,J,K) :- A=L, B=L, C=true, D=F, E=M, G=N, H=L, I=true,
                 J=0, O=true, P=Q, Q-L=< -1, F>=0, Q-F>=0, new6(A,L,O,P,M,Q,N,H,I,J,K).
19. new7(A,B,C,D,E,F) :- A=C, B=true, D=true, E=0, F=true, C>=0.
20. new7(A,B,C,D,E,F) :- A=G, B=H, C=G, D=true, E=0, F=H, I=true, G>=0, J-G>=0,
                 new10(G,I,J,H).
21. new9(A,B,C,D,E,F,G) :- A=D, B=D, C=true, E=true, F=0, G=true, D>=1.
22. new9(A,B,C,D,E,F,G) :- A=H, B=H, C=true, D=H, E=true, F=0, G=I, J=true, H>=1,
        K>=H, new10(H,J,K,I).
23. new9(A,B,C,D,E,F,G) :- A=H, B=H, C=true, D=H, E=true, F=0, I=true, H>=1,
                 new9(A,H,I,D,E,F,G).
24. new10(A,B,C,D) :- B=true, D=true, A>=0, C-A>=0.
25. new10(A,B,C,D) :- A=E, B=true, D=false, F=true, E-C=< -1, E>=0, new11(E,F).
26. new10(A,B,C,D) :- A=E, B=true, D=F, G=true, E-C=<0, E>=0, H>=C, new10(E,G,H,F).
27. new11(A,B) :- B=true, A>=0.
28. new11(A,B) :- A=C, B=true, D=true, C>=0, new11(C,D).
```

[2]The tool is available at https://fmlab.unich.it/iclp2018/.