Proving Properties of Sorting Programs: A Case Study in Horn Clause Verification

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

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

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

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

The proof of a program property can be reduced to the proof of satisfiability of a set of constrained Horn clauses (CHCs) which can be automatically generated from the program and the property. In this paper we have conducted a case study in Horn clause verification by considering several sorting programs with the aim of exploring the effectiveness of a transformation technique which allows us to eliminate inductive data structures such as lists or trees. If this technique is successful, we derive a set of CHCs with constraints over the integers and booleans only, and the satisfiability check can often be performed in an effective way by using state-of-the-art CHC solvers, such as Eldarica or Z3. In this case study we have also illustrated the usefulness of a companion technique based on the introduction of the so-called *difference predicates*, whose definitions correspond to lemmata required during the verification. We have considered functional programs which implement the following kinds of sorting algorithms acting on lists of integers: (i) linearly recursive sorting algorithms, such as insertion sort and selection sort, and (ii) non-linearly recursive sorting algorithms, such as quicksort and mergesort, and we have considered the following properties: (i) the partial correctness properties, that is, the orderedness of the output lists, and the equality of the input and output lists when viewed as multisets, and (ii) some arithmetic properties, such as the equality of the sum of the elements before and after sorting.

1 Introduction

Recent work has shown that *constrained Horn clauses* (CHCs) provide a suitable basis for program verification [1], and indeed many program verification problems can be reduced in a very natural way to CHC satisfiability problems. This fact has motivated the development of a number of tools, called *CHC solvers*, which can perform satisfiability proofs. Most of them work well on CHCs with simple constraint theories, such as the theory of Linear Integer Arithmetic (LIA) and the theory of Booleans (Bool). Among these tools we mention: Eldarica [14], FreqHorn [11], HSF [12], RAHFT [15], Spacer [16], VeriMAP [4], and Z3 [19].

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

E. De Angelis, G. Fedyukovich, N. Tzevelekos and M. Ulbrich (Eds.):
Sixth Workshop on Horn Clauses for Verification and Synthesis and
Third Workshop on Program Equivalence and Relational Reasoning.
EPTCS 296, 2019, pp. 48–75, doi:10.4204/EPTCS.296.8

In order to overcome this difficulty, the following two approaches have recently been suggested. (i) The first one consists in the incorporation into CHC solvers of suitable induction principles [20, 22], and (ii) the second one consists in transforming the given set of CHCs into a new set where inductively defined data structures are removed, and whose satisfiability implies the satisfiability of the original clauses [8, 18].

In this paper we will follow this second approach and, in particular, we will consider the *Elimination Algorithm* presented in a previous work of ours [8], which implements a transformation strategy for removing inductively defined data structures and it is based on the familiar fold/unfold rules [10, 21]. Thus, if the clauses derived by the Elimination Algorithm have all their constraints in the LIA or Bool theory, then there is no need to modify the CHC solvers for performing the required satisfiability proofs.

As usual in the transformation-based approach, the success of the Elimination Algorithm depends on the introduction of some suitable auxiliary predicates.

The novel contribution of this paper is the study of various verification problems of sorting programs which we have verified by using a technique for introducing auxiliary predicates, called *the introduction of difference predicates* [9]. The name of the difference predicates comes from the fact that they express the relation between the values computed by two different functions. The definitions of those predicates also correspond to the lemmata required for the proofs of the properties of interest by structural induction.

By extending the Elimination Algorithm with the introduction of difference predicates, we can remove inductively defined data structures in cases where the plain Elimination Algorithm would not terminate. By doing so we extend the effectiveness of state-of-the-art CHC solvers when proving program properties.

In Sections 2 and 3, we will consider various verification problems of sorting algorithms written as functional programs acting on lists of integers. In particular in Section 2 we deal with linearly recursive sorting algorithms, while in Section 3 we deal with non-linearly recursive ones. The verification is done into two phases. In Phase (i) we derive by transformation, introducing suitable difference predicates, a set of CHCs on LIA and Bool constraints only (that is, constraints on lists will no longer be present in the derived clauses), and then in Phase (ii) we prove the satisfiability of the derived CHCs by using the solver Eldarica acting on LIA and Bool constraints. We have also proved that Eldarica is not able (within a given time-out) to check satisfiability of the clauses which are obtained by the direct translation into CHCs of the given functional programs and properties, before the transformation performed at Phase (i). In Section 4 we present the related work and we make some concluding remarks. In the Appendix we provide the technical details of the verifications we have performed by using our verification and transformation system VeriMAP [4] and the CHC solver Eldarica [14].

2 Verification of Linearly Recursive Sorting Algorithms

In this section, we consider two functional programs implementing simple linearly recursive sorting algorithms: *InsertionSort* and *SelectionSort*. Then, we verify various properties of those programs by applying our transformation-based method. The transformations presented here and in the next section are done by an interactive system [1]. Their mechanization is briefly discussed in Section 4.

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

[1]The MAP Transformation System `http://www.iasi.cnr.it/~proietti/system.html`

```
type list = Nil | Cons of int * list;;
let rec ins x l =
  match l with
  | Nil -> Cons(x,Nil)
  | Cons(y,ys) -> if x<=y then Cons(x,Cons(y,ys)) else Cons(y,ins x ys);;
let rec insertionSort l =
  match l with
  | Nil -> Nil
  | Cons(x,xs) -> ins x (insertionSort xs);;
let rec count x l =
  match l with
  | Nil -> 0
  | Cons(y,ys) -> if x=y then 1 + count x ys else count x ys;;
```

where: (i) given an integer `x` and a list `l` of integers, ordered in the ascending order, `ins x l` returns the ordered list made out of `x` and the integers in `l`, (ii) `insertionSort l` returns an ordered permutation of the integers in the list `l`, and (iii) `count x l` returns the number of occurrences of the integer `x` in the (not necessarily ordered) list `l`.

Let us suppose that we want to prove the following property of *InsertionSort* :

$\forall \texttt{l},\texttt{s},\texttt{x}.\ (\texttt{insertionSort l}) = \texttt{s} \ \rightarrow\ (\texttt{count x l}) = (\texttt{count x s})$ (*IS_Perm*)

which states that the list `s` returned by `insertionSort l` is a permutation of the given list `l`.

In order to prove this property, we first consider the translation of *InsertionSort* and *IS_Perm* into the following set of constrained Horn clauses (we will not enter into the details of this translation here and the reader may refer to [8, 22]) [2]:

```
1. false :- N1=\=N2, insertionSort(L,S), count(X,L,N1), count(X,S,N2).
2. ins(I,[],[I]).
3. ins(I,[X|Xs],[I,X|Xs]) :- I=<X.
4. ins(I,[X|Xs],[X|Ys]) :- I>X, ins(I,Xs,Ys).
5. insertionSort([],[]).
6. insertionSort([X|Xs],S) :- insertionSort(Xs,S1), ins(X,S1,S).
7. count(X,[],0).
8. count(X,[H|T],N) :- X=H,  N=M+1, count(X,T,M).
9. count(X,[H|T],N) :- X=\=H, count(X,T,N).
```

The translation ensures that (i) `ins(X,L,L1)`, (ii) `insertionSort(L,S)`, and (iii) `count(X,L,N)` hold iff (i) `ins X L = L1`, (ii) `insertionSort L = S`, and (iii) `count X L = N`, respectively, hold in the program *InsertionSort*. Clause `1` translates property *IS_Perm* as it stands (using the functional notation) for:

$$\forall \texttt{l},\texttt{s},\texttt{x},\texttt{n1},\texttt{n2}.\ (\texttt{insertionSort l}) = \texttt{s} \wedge (\texttt{count x l}) = \texttt{n1} \wedge (\texttt{count x s}) = \texttt{n2} \ \rightarrow\ \texttt{n1} = \texttt{n2}$$

Now, as it is well-known from the literature, we have that clauses `1`–`9` are satisfiable iff *IS_Perm* holds. Unfortunately, state-of-the-art CHC solvers, such as Eldarica or Z3, fail to prove satisfiability of those clauses, because these solvers do not incorporate any induction principle on lists.

Thus, we proceed by applying a transformation technique based on the Elimination Algorithm [8] whose objective is to eliminate the inductively defined data structures and, in particular, in our case, the

[2] When writing clauses we use a Prolog-like syntax, instead of the more verbose SMT-LIB syntax.

lists. That algorithm is successful only if it is combined with a companion technique for the introduction of the so-called *difference predicates* whose application we will now illustrate.

At the end of the transformation, if the derived clauses have no occurrences of lists, we can check their satisfiability by using a state-of-the-art CHC solver on LIA. From the proof of satisfiability, we will then get the proof of the property *IS_Perm* because during the transformation every rule that has been applied preserves satisfiability [10].

Let us start off by applying the Elimination Algorithm to clause `1`. We introduce a new predicate `new1` defined by the following clause (here and in what follows the *n* atoms in the body of clause `m` are identified by numbers `m.1`, ..., `m.n`, respectively):

```
10. new1(X,N1,N2) :- insertionSort(L,S), count(X,L,N1), count(X,S,N2).
                          (10.1)               (10.2)          (10.3)
```

The atoms in the body of clause `10` are the atoms in the body of clause `1` and the arguments of `new1` are the non-list variables occurring in those atoms (this choice is consistent with our objective of eliminating the list variables). We fold clause `1` using clause `10` and we get the following clause with no list variables:

```
11. false :- N1=\=N2, new1(X,N1,N2).
```

Now we proceed by eliminating lists from the newly introduced clause `10`. We unfold it and we get:

```
12. new1(X,0,0).
13. new1(X,N1,N2) :- N1=N+1, insertionSort(Xs,S1), ins(X,S1,S),
                                  (13.1)               (13.2)
                     count(X,Xs,N), count(X,S,N2).
                        (13.3)        (13.4)
14. new1(X,N1,N2) :- X=\=Y, insertionSort(Xs,S1), ins(Y,S1,S),
                     count(X,Xs,N1), count(X,S,N2).
```

At this point it is impossible to fold clauses `13` and `14` using clause `10` because of the mismatch of the variables `S` and `S1`. Thus, we apply the difference predicate technique which works according to the following Steps 1–6. Let us consider clause `13`. Analogous steps can be performed for clause `14` and we leave them to the reader.

• *Step* 1. *Embed*. Each of the atoms `13.1`, `13.3`, and `13.4` of clause `13` to be folded is a variant of atom `10.1`, `10.2` and `10.3`, respectively, of clause `10` to be used for folding. We say that clause `10` *is embedded into* clause `13`. However, their conjunction is *not* a variant of the body of clause `10`, and thus we cannot fold clause `13` using clause `10`.

• *Step* 2. *Rename*. We rename apart clause `10` to be used for folding, so as to have fresh, new variable names that do not occur anywhere else. This renaming apart will avoid clashes of variables in all the subsequent steps. We get:

```
10a. new1(Xa,N1a,N2a) :- insertionSort(La,Sa), count(Xa,La,N1a), count(Xa,Sa,N2a).
                              (10a.1)               (10a.2)           (10a.3)
```

• *Step* 3. *Match*. We match the body of clause `10a` against the body of clause `13` to be folded. We manage to match the conjunction (`10a.1`, `10a.2`) with the conjunction (`13.1`, `13.3`) by the renaming substitution $\sigma = \{$`La/Xs`, `Sa/S1`, `Xa/X`, `N1a/N`$\}$, but we cannot extend this matching to the remaining atoms `10a.3` and `13.4` because the substitution $\{$`Xa/X`, `Sa/S`, `N2a/N2`$\}$ is inconsistent with σ. By applying the substitution σ, we get the following clause `10m`, which is a variant of clause `10a`:

```
10m. new1(X,N,N2a) :- insertionSort(Xs,S1), count(X,Xs,N), || count(X,S1,N2a).
                           (10m.1)             (10m.2)          (10m.3)
```

This clause `10m` is the actual clause which we will use for folding at Step 6 below. The marker || in the body of clause `10m` has no logical meaning and it is only used for separating the *matching conjunction* (atoms `10m.1` and `10m.2`) to its left and the *mismatching conjunction* (atom `10m.3`) to its right. In general, also the mismatching conjunction may consist of more than one atom, although in our case it is made out of one atom only. Also for the clause to be folded (clause `13` in our case) we define the matching and the mismatching conjunctions: (i) the *matching conjunction* is equal to the one of the clause we will use for folding (atoms `13.1` and `13.3` in our case), while (ii) the *mismatching conjunction* is made out of all body atoms of the clause to be folded that do not belong to the matching conjunction (atoms `13.2` and `13.4` in our case).

• *Step* 4. *Introduce a Difference Predicate*. Now, in order to fold clause `13` using clause `10m` we need to replace the mismatching conjunction of clause `13` by the mismatching conjunction of clause `10m`. This replacement can be done at the expense of adding to the body of clause `13` a new atom with a so-called *difference predicate*. This atom addition is required for preserving satisfiability of the derived clauses. In our case the difference predicate we introduce, called `diff1`, is defined as follows:

```
15. diff1(X,N2a,N2) :- ins(X,S1,S), count(X,S,N2), †† count(X,S1,N2a).
                       (13.2)       (13.4)            (10m.3)
```

This definition clause is constructed as we now specify. The body is made out of two conjunctions separated by the marker ††: (i) the first one to the left of †† is the mismatching conjunction of the clause to be folded (atoms `13.2` and `13.4` in our case), and (ii) the second conjunction to the right of †† is the mismatching conjunction of the clause we will use for folding (atom `10m.3` in our case). The arguments `X`, `N2a`, and `N2` of the predicate `diff1` are the non-list variables occurring in the body we have just constructed (obviously these arguments can be placed in any order). The marker †† in clause `15` separates the atoms to its left that should be removed from the body of clause `13` from the atoms to its right that should be added to the body of clause `13` so that folding of clause `13` can be performed as we desire (see Step 6 below).

The reader may note that the difference predicate `diff1(X,N2a,N2)` expresses the relation between the non-list output variable `N2a` of the atom that is added and the non-list output variable `N2` of the atoms that are removed [3].

• *Step* 5. *Replace*. In the clause to be folded (in our case clause `13`) we replace the mismatching conjunction (in our case atoms `13.2` and `13.4`) by: (i) the mismatching conjunction of the clause we will use for folding (that is, atom `10m.3`), and (ii) the head of the definition of the difference predicate `diff1`. Note that no extra variable renamings are required besides that of Step (2). We get the following clause:

```
13r. new1(X,N1,N2) :- N1=N+1, insertionSort(Xs,S1), count(X,Xs,N),
                                    (13.1)              (13.3)
                           count(X,S1,N2a), diff1(X,N2a,N2).
                              (10m.3)
```

• *Step* 6. *Fold*. We fold clause `13r` using clause `10m` (this folding is possible by construction) and we get:

```
13f. new1(X,N1,N2) :- N1=N+1, new1(X,N,N2a), diff1(X,N2a,N2).
```

It can be shown that the above Steps 1–6 preserve satisfiability of clauses, and this is an essential requirement in our proof of the property *IS_Perm*. A detailed proof of this fact is outside the scope of the present paper.

[3] The input and output variables of the atoms `10m.3`, `13.2`, and `13.4` are defined as expected, if one considers the associated functional expressions `(count X S1)`, `(ins X S1)`, and `(count X S)`, respectively.

Analogous steps can be performed starting from clause `14`, and during these steps we introduce a new difference predicate `diff2` defined by the following clause:

```
16. diff2(X,Y,N2b,N2) :- X=\=Y, ins(X,S1,S), count(Y,S,N2), count(Y,S1,N2b).
```

Note that, unlike clause `15` defining the predicate `diff1`, clause `16` has in its body also the constraint `X=\=Y`. We have to add this constraint because: (i) it occurs in the body of the clause to be folded (clause `14`, in our case), and (ii) it relates variables (`X` and `Y`, in our case) occurring in the body atoms.

The reader may check that, as from clause `13` we derived clause `13f`, from clause `14` we may derive the following folded clause:

```
14f. new1(X,N1,N2) :- X=\=Y, new1(Y,N1,N2b), diff2(X,Y,N2b,N2).
```

The CHCs we have obtained so far are: `11`, `12`, `13f`, `14f`, together with: (i) clauses `15` and `16` defining the two difference predicates `diff1` and `diff2`, respectively, and (ii) clauses `2`–`4` and `7`–`9` defining the predicates `ins` and `count`. Since clauses `15` and `16` still have list variables in their bodies, we continue the application of the Elimination Algorithm [8] with the objective of eliminating list variables. During this process, which we will not describe here, no new difference predicates need to be introduced, and we get the following final set of clauses, besides clauses `11`, `12`, `13f`, `14f`:

```
diff1(X,0,N2)  :- N2=N1+1, new2(X,N1).
diff1(X,N1,N2) :- N2=M2+1, N1=M1+1, new3(X,M2,M1),
diff1(X,N1,N2) :- X=<Y, N2=N+1, X=\=Y, new4(X,Y,N,N1).
diff2(X,Y,0,0) :- Y=\=X.
diff2(X,Y,M,N) :- X=<Y, Y=\=X, M=K+1, new3(Y,N,K).
diff2(X,Y,M,N) :- X=<Z, Y=\=X, Y=\=Z, N=M, new5(Y,N).
diff2(X,Y,M,N) :- X>Y, N=H+1, M=K+1, diff2(X,Y,K,H).
new2(X,0).
new3(X,N1,N) :- N1=N+1, new5(X,N).
new4(X,Y,N,N) :- X=<Y, X=\=Y, new5(X,N).
new5(X,0).
new5(X,N1) :- N1=N+1, new5(X,N).
```

If we submit this final set of clauses to the CHC solver Eldarica [13], it is able to show its satisfiability by constructing the following model (we present it using Eldarica's *Prolog format*):

```
diff1(A,B,C) :- (((B - C) = -1), (B >= 0)).
diff2(A,B,C,D) :- ((C = D), (C >= 0)).
new1(A,B,C) :- ((B = C), (B >= 0)).
new2(A,B) :- (B = 0).
new3(A,B,C) :- (((C - B) = -1), (B >= 1)).
new4(A,B,C,D) :- ((D = C), ((C >= 0), ((B - A) >= 1))).
new5(A,B) :- (B >= 0).
```

The existence of this model shows that property *IS_Perm* holds for program *InsertionSort*.

As already mentioned, the name 'difference predicates' comes from the fact that they express a relation between the values of different functional expressions. For instance, the atom `diff1(X,N2a,N2)` expresses the relation between the value `N2a` of the expression `(count X S1)` and the value `N2` of the expression `(count X (ins X S1))`.

Actually, one can also show that there is a tight correspondence between the definition of the difference predicates and the lemmata which are needed for making the inductive proofs of the properties to be shown. In particular, in the case of property *IS_Perm*, if we replace in clause `15` defining diff1(X,N2a,N2) its model computed by Eldarica in the process of showing the satisfiability of the final set of clauses, that is, N2 = N2a + 1 ∧ N2a >= 0, from clause 15 we get the following lemma which is required for the proof of *IS_Perm* by induction on the list structure, namely,

$$\forall\, \mathtt{X}, \mathtt{S1}, \mathtt{S}, \mathtt{N2}, \mathtt{N2a}.\ \mathtt{ins(X,S1,S)} \wedge \mathtt{count(X,S,N2)} \wedge \mathtt{count(X,S1,N2a)} \rightarrow (\mathtt{N2} = \mathtt{N2a} + 1 \wedge \mathtt{N2a} >= 0)$$

Similarly to the above proof of property *IS_Perm*, one can prove that program *InsertionSort* also satisfies the following property:

$$\forall \mathtt{l}, \mathtt{s}.\ (\mathtt{insertionSort\ l}) = \mathtt{s} \ \rightarrow\ (\mathtt{ordered\ s}) \qquad (\textit{IS\_Orderedness})$$

where the function ordered from integer lists to booleans is defined as follows:

```
let rec ordered l =
  match l with
  | Nil -> true
  | Cons(x,xs) -> match xs with
                  | Nil -> true
                  | Cons(y,ys) -> x<=y & (ordered xs);;
```

We start from the initial clauses 2–9 together with the following clause which translates the property *IS_Orderedness*:

```
false :- B=false, insertionSort(L,S), ordered(S,B).
```

and after a transformation similar to the one we have described for property *IS_Perm*, we get the following final set of clauses:

```
false :- B=false, new1(B).
new1(true).
new1(B) :- new1(B1), diff(X,B,B1).
diff(I,true,true).
diff(I,B,B).
diff(I,false,false).
diff(I,true,true).
diff(I,B,B1) :- new2(I,Y1,Y,B,B1).
new2(I,Y1,Y,B,B) :- I=<Y1, I=Y.
new2(I,Y1,Y,false,false) :- I>Y, Y1=Y.
new2(I,Y1,Y,true,true) :- I>Y, Y1=Y.
new2(I,Y1,Y,B,B1) :- I>Y, Y=<Y2, Y=<Y3, Y1=Y, new2(I,Y3,Y2,B,B1).
```

whose satisfiability can be shown by Eldarica. The model that Eldarica finds is the following:

```
new1(A) :- (A = true).
diff(A,B,C) :- (\+((C = true)); (B = true)).
new2(A,B,C,D,E) :- (\+((E = true)); (D = true)).
```

Having shown the properties *IS_Perm* and *IS_Orderedness* the automatic proof of partial correctness of program *InsertionSort* is completed.

We have also proved various other properties of linearly recursive sorting programs using the techniques we have presented above based on the application of the Elimination Algorithm and the introduction of difference predicates.

In particular, we have shown that *InsertionSort* satisfies the following properties:

(i) $\forall$ `l,s.` `(insertionSort l) = s` $\rightarrow$ `(length l) = (length s)` (*IS_Length*)

(ii) $\forall$ `l,s.` `(insertionSort l) = s` $\rightarrow$ `(sumlist l) = (sumlist s)` (*IS_Sum*)

where `length l` computes the numbers of elements in the list `l`, and `sumlist l` computes the sum of the elements in the integer list `l`.

Moreover, for the following *SelectionSort* program:

```
let rec min (Cons(x,xs)) =
   match xs with
   | Nil -> x
   | Cons(y,ys) -> if x<y then min (Cons(x,ys)) else min (Cons(y,ys));;
let rec delete m (Cons(x,xs)) =
   if m=x then xs else Cons(x,delete m xs);;
let rec selectionSort l =
   match l with
   | Nil -> Nil
   | Cons(x,xs) -> let m = min (Cons(x,xs)) in
                        (Cons(m,selectionSort(delete m (Cons(x,xs)))));;
```

we have shown the following three properties:

(i) $\forall$ `l,s,x.` `(selectionSort l)=s` $\rightarrow$ `(count x l)=(count x s)` (*SS_Perm*)

(ii) $\forall$ `l,s.` `(selectionSort l)=s` $\rightarrow$ `(ordered s)` (*SS_Orderedness*)

(iii) $\forall$ `l,s.` `(selectionSort l)=s` $\rightarrow$ `(length l)=(length s)` (*SS_Length*)

3 Verification of Non-Linearly Recursive Sorting Algorithms

Now, we consider two popular non-linearly recursive sorting algorithms: *QuickSort* and *MergeSort*, and we show how our transformation-based method allows us to verify some properties of those algorithms.

Let us first consider the following *QuickSort* program which implements a simplified version of the QuickSort algorithm:

```
let rec partition x l =
  match l with
  | Nil -> (Nil,Nil)
  | Cons(y,ys) -> let (l1,l2) = partition x ys in
                  if x>=y then (Cons(y,l1),l2) else (l1,Cons(y,l2));;
let rec append l ys =
  match l with
  | Nil -> ys
  | Cons(x,xs) -> Cons(x, append xs ys);;
let rec quickSort l =
  match l with
  | Nil -> Nil
  | Cons(x,xs) -> let (ys,zs) = partition x xs in
                  append (quickSort ys) (Cons(x,(quickSort zs)));;
```

where, given an integer `x` and an integer list `l`, `partition x l` returns a pair `(l1,l2)` of lists such that their concatenation is `l` itself and all elements of `l1` are all the elements of `l` not larger than `x`. Functions `append` and `quickSort` behave as expected.

Similarly to the examples of Section 2 we want to prove the following property of *QuickSort* :

$\forall$`l,s,x. (quickSort l)=s` $\rightarrow$ `(count x l)=(count x s)` (*QS_Perm*)

which states that the list `s` returned by `quickSort l` is a permutation of the list `l`.

Program *QuickSort* and property *QS_Perm* are translated into the following constrained Horn clauses:

```
1. false :- M=\=N, quickSort(L,S), count(X,L,M), count(X,S,N).
2. partition(X,[],[],[]).
3. partition(X,[Y|Ys],[Y|L1],L2) :- Y=<X, partition(X,Ys,L1,L2).
4. partition(X,[Y|Ys],L1,[Y|L2]) :- X<Y, partition(X,Ys,L1,L2).
5. append([],Ys,Ys).
6. append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
7. quickSort([],[]).
8. quickSort([X|Xs],S) :- partition(X,Xs,Ys,Zs),
      quickSort(Ys,S1), quickSort(Zs,S2), append(S1,[X|S2],S).
```

The clauses for `count` are as in Section 2. Now, we transform the set made out of clauses 1–8, together with the clauses for `count`, into a new set of clauses without lists. We start off by introducing a new predicate `new1` by the following clause (note that the arguments of `new1` are the non-list variables occurring in the body):

```
9. new1(X,M,N) :- quickSort(L,S), count(X,L,M), count(X,S,N).
```

By folding clause `1`, we derive a new clause without occurrences of lists:

```
10. false :- M=\=N, new1(X,M,N).
```

We proceed by unfolding clause `9` and we derive the following clauses, where we have rearranged the order of the atoms in the bodies to simplify the explanation of the subsequent steps:

```
11. new1(X,0,0).
12. new1(X,M,N) :- M=K+1, X=Y,
       quickSort(L1,S1), quickSort(L2,S2), count(X,L,K), partition(Y,L,L1,L2),
       append(S1,[Y|S2],S), count(X,S,N).
13. new1(X,M,N) :- X=\=Y,
       quickSort(L1,S1), quickSort(L2,S2), count(X,L,M), partition(Y,L,L1,L2),
       append(S1,[Y|S2],S), count(X,S,N).
```

It is impossible to fold clauses `12` and `13` using clause `9`, and thus, similarly to the *InsertSort* example in Section 2, we apply the difference predicate technique. However, the presence of the two recursive calls to `quickSort`, makes things a bit more involved in this case. We show the derivation relative to clause `12`. The one relative to clause `13` is similar and we omit it.

• *Step* 1. *Embed*. The atoms `quickSort(L1,S1)`, `count(X,L,K)`, `count(X,S,N)` in the body of clause `12` are individually variants of atoms in the body of clause `9`. However, their conjunction is *not* a variant of the body of clause `9`, and thus we cannot fold clause `12` using clause `9`. Similarly, `quickSort(L2,S2)`, `count(X,L,K)`, `count(X,S,N)` is not a variant of the body of clause `9`.

• *Step* 2. *Rename*. In order to fold the two `quickSort` atoms in the body of clause `12` we consider two renamed apart variants of clause `9`. We get:

```
9a. new1(Xa,Ma,Na) :- quickSort(La,Sa), count(Xa,La,Ma), count(Xa,Sa,Na).
9b. new1(Xb,Mb,Nb) :- quickSort(Lb,Sb), count(Xb,Lb,Mb), count(Xb,Sb,Nb).
```

• *Step* 3. *Match.* We simultaneously match clauses 9a and 9b against clause 12 to be folded. We get the substitution $\sigma = \{$La/L1, Sa/S1, Xa/X, Lb/L2, Sb/S2, Xb/X$\}$. This substitution cannot be extended so to include more bindings obtained by pairs of count atoms, because: (i) we cannot add the bindings La/L, Sa/S, Lb/L, Sb/S, which are inconsistent with the bindings for the same variables already in σ, and (ii) we cannot add bindings for the output variables Ma, Mb, Na, Na, because we stipulate that the output variables of two predicates may match only if all their input variables match [4]. For instance, as already said, the matching of the input variables La and L of the atoms count(Xa,La,Ma) and count(X,L,M) fails, and hence we cannot add the binding Ma/M for their output variables.

By applying σ we get the following two instances of clauses 9a and 9b:

```
9am. new1(X,Ma,Na) :- quickSort(L1,S1), || count(X,L1,Ma), count(X,S1,Na).
9bm. new1(X,Mb,Nb) :- quickSort(L2,S2), || count(X,L2,Mb), count(X,S2,Nb).
```

These clauses are the two clauses we will actually use for folding clause 12. Thus, the only atoms that are made equal by σ are the two quickSort atoms in the bodies of clauses 9am and 9bm (which we place to the left of the marker ||), and the left and right quickSort atoms in the body of clause 12, respectively. As in clause 10m of Section 2, the marker || separates the matching conjunction to its left from the mismatching conjunction to its right.

The mismatching conjunctions of clauses 9am and 9bm we will use for folding (which we place to the right of the markers ||) are made out of the atoms count(X,L1,Ma), count(X,S1,Na), and count(X,L2,Mb), count(X,S2,Nb), respectively, while the mismatching conjunction of clause 12 to be folded is made out of the following atoms:

```
count(X,L,K), partition(Y,L,L1,L2), append(S1,[Y|S2],S), count(X,S,N).    (*)
```

• *Step* 4. *Introduce Difference Predicates.* Now, in order to fold clause 12 using the two clauses 9am and 9bm, we need to replace the mismatching conjunction of clause 12 by the mismatching conjunctions of clauses 9am and 9bm. This replacement can be done at the expense of adding to the body of clause 12 two new atoms with *difference predicates*. As already remarked, this atom addition is required for preserving satisfiability of the derived clauses. In our case the difference predicates we introduce, called diff1 and diff2, are defined as follows:

```
14.  diff1(X,Y,K,Ma,Mb) :- X=Y, count(X,L,K), partition(Y,L,L1,L2), ††
                           count(X,L1,Ma), count(X,L2,Mb).
15.  diff2(X,Y,N,Na,Nb) :- X=Y, append(S1,[Y|S2],S), count(X,S,N), ††
                           count(X,S1,Na), count(X,S2,Nb).
```

These two clauses are constructed as we now specify. (i) We first consider the set of atoms occurring in the mismatching conjunction of the clause to be folded (in our case the atoms (*) of clause 12) and in the mismatching conjunctions of the clauses we will use for folding (in our case the atoms to the right of || in clauses 9am and 9bm), and then we partition that set into two subsets by keeping together only the atoms that share common list variables. These two subsets constitute the bodies of the clauses 14 and 15. (ii) We add the constraint X=Y to the bodies of clause 14 and 15 to define more specific predicates as needed in the clause 12 to be folded, where the same constraint occurs. (iii) We specify the arguments of diff1 and diff2 to be the non-list variables of the corresponding bodies (obviously, the order of the arguments is insignificant).

The markers †† in clauses 14 and 15 separate the atoms to be removed from the body of clause 12 (those to the left) from the atoms to be added to the body of clause 12 (those to the right) so that the

[4]The input and output variables of the predicate count are defined as expected, if one considers the function count of the functional program it comes from.

folding steps can then be performed.

The reader may note that the difference predicates express the relations between the non-list output variables of the atoms that are added and the non-list output variables of the atoms that are removed. In particular, for clause 14, the atom diff1(X,Y,K,Ma,Mb) expresses the relation between the values of the output variables K, Ma, and Mb of the count atoms, for all values of the input variables X and Y of the count atoms. (The output variables L1 and L2 of the removed atom partition(Y,L,L1,L2) do not play any role because they are list variables.)

Analogously, for clause 15, the atom diff2(X,Y,N,Na,Nb) expresses the relation between the values of the output variables N, Na, and Nb of the count atoms, for all values of the input variables X and Y of the count atoms.

Note that we could have introduced, instead of two difference predicates, a single difference predicate by a clause whose body was the conjunction of the atoms in the bodies of clauses 14 and 15. Thus, that conjunction would have been made out of exactly the mismatching conjunctions of the clause to be folded (clause 12) and the clauses used for folding (clauses 9am and 9bm), as in the case of the *InsertSort* example in Section 2. Also this single, more complex difference predicate would have allowed the elimination of lists.

• *Step* 5. *Replace*. In clause 12 we replace the atoms to the left of †† in the bodies of clauses 14 and 15, by the atoms to the right of ††, and we also add the heads of clauses 14 and 15. We get (after a reordering of the atoms):

```
16. new1(X,M,N) :- M=K+1, X=Y, diff1(X,Y,K,Ma,Mb),
          quickSort(L1,S1), count(X,L1,Ma), count(X,S1,Na),
          quickSort(L2,S2), count(X,L2,Mb), count(X,S2,Nb), diff2(X,Y,N,Na,Nb).
```

It can be shown that, if the values of K and N which are arguments of the difference predicates diff1 and diff2, are functions of (X,Y,Ma,Mb) and (X,Y,Na,Nb), respectively, then the clauses before this replacement are satisfiable *if and only if* the clauses after the replacement are satisfiable. The proof of this fact is outside the scope of this paper.

• *Step* 6. *Fold*. Now, we are able to fold twice clause 16 using clauses 9am and 9bm, and we derive the following clause, where all arguments are integer variables:

```
17. new1(X,M,N) :- M=K+1, X=Y, diff1(X,Y,K,Ma,Mb), new1(X,Ma,Na),
                    new1(X,Mb,Nb), diff2(X,Y,N,Na,Nb).
```

Now, as the reader may check, if we start from clause 13 and we perform in a similar way the six steps we have performed starting from clause 12 for deriving clause 17, we get the following clause:

```
18. new1(X,M,N) :- X=\=Y, diff3(X,Y,M,Ma,Mb), new1(X,Ma,Na),
                    new1(X,Mb,Nb), diff4(X,Y,N,Na,Nb).
```

where diff3 and diff4 are difference predicates defined by the following clauses:

```
19. diff3(X,Y,M,Ma,Mb) :- X=\=Y, count(X,L,M), partition(Y,L,L1,L2),
                           count(X,L1,Ma), count(X,L2,Mb).
20. diff4(X,Y,N,Na,Nb) :- X=\=Y, append(S1,[Y|S2],S), count(X,S,N),
                           count(X,S1,Na), count(X,S2,Nb).
```

At this point we can derive clauses without lists for diff1–diff4 by applying the Elimination Algorithm [8]. Indeed, by introducing some new predicates, namely new2, new3, and new4, we get:

```
21. diff1(X,Y,0,0,0)   :- X=Y.
22. diff1(X,Y,M,Ma,Mb) :- X=Y, Ma=Ma1+1, M=M1+1, diff1(X,Y,M1,Ma1,Mb).
23. diff2(X,Y,N,0,Nb)  :- X=Y, N=Nb+1, new2(X,Nb).
24. diff2(X,Y,N,Na,Nb) :- X=Y, Na=Na1+1, N=N1+1, diff2(X,Y,N1,Na1,Nb).
25. diff3(X,Y,M,Ma,Mb) :- X<Y, new3(X,Y,M,Ma,Mb).
26. diff3(X,Y,M,Ma,Mb) :- X>Y, new4(X,Y,M,Ma,Mb).
27. diff4(X,Y,N,0,N)   :- X=\=Y, new2(X,N).
28. diff4(X,Y,N,Na,Nb) :- X=\=Y, Na=Na1+1, N=N1+1, diff4(X,Y,N1,Na1,Nb).
29. new2(X,0).
30. new2(X,N) :- N=N1+1, new2(X,N1).
31. new3(X,Y,0,0,0)   :- X<Y.
32. new3(X,Y,M,Ma,Mb) :- X<Y, M=M1+1, Ma=Ma1+1, new3(X,Y,M1,Ma1,Mb).
33. new4(X,Y,0,0,0)   :- X>Y.
34. new4(X,Y,M,Ma,Mb) :- X>Y, M=M1+1, Mb=Mb1+1, new4(X,Y,M1,Ma,Mb1).
```

The final set of clauses without list arguments consists of clauses 10, 11, 17, 18, 21–34. Eldarica is able to prove the satisfiability of this set of clauses by computing a model, which is expressible in LIA (see Appendix). Thus, we have proved that property *QS_Perm* holds for the *QuickSort* program.

For *QuickSort* we have also proved the following arithmetic property by using the same technique shown above:

$$\forall \mathtt{l}, \mathtt{s}.\ (\mathtt{quickSort}\ \mathtt{l}) = \mathtt{s} \rightarrow (\mathtt{sumlist}\ \mathtt{l}) = (\mathtt{sumlist}\ \mathtt{s}) \qquad (\textit{QS\_Sum})$$

Now as a second example of non-linearly recursive sorting programs, let us now consider the following *MergeSort* program:

```
let rec takedrop d l =
   match l with
   | Nil -> (Nil,Nil)
   | Cons(x,xs) -> if d=0 then (Nil,l) else
                   let (a,b) = takedrop (d-1) xs in (Cons(x,a),b);;
let rec split l =
   let d = (length l)/2 in takedrop d l;;

let rec merge l1 l2 =
   match l1 with
   | Nil -> l2
   | Cons(x,xs) -> match l2 with
                   | Nil -> l1
                   | Cons(y,ys) -> if x<y then Cons(x,merge xs l2) else
                           Cons(y, merge l1 ys);;

let rec mergeSort l =
   match l with
   | Nil -> Nil
   | Cons(x,xs) ->
         match xs with
         | Nil -> Cons(x,Nil)
         | Cons(y,ys) -> let (fst,snd) = split l in
                            merge (mergeSort fst) (mergeSort snd);;
```

where, given an integer `d` and a list `l`, `takedrop d l` returns a pair `(fst,snd)` of lists such that: (i) `fst` is a list made out of the first `d` elements of `l`, and (ii) `snd` is a list made out of the remaining elements of `l`. The other functions of *MergeSort* behave as expected. Note that the value of `d` in the function call `split l` is obtained by integer division, and thus `d` is the largest integer smaller or equal to the length of `l` divided by 2.

For *MergeSort* we have proved the following property:

$$\forall \mathtt{l},\mathtt{s}.\ (\mathtt{mergeSort\,l}) = \mathtt{s} \rightarrow (\mathtt{sumlist\,l}) = (\mathtt{sumlist\,s}) \qquad (\mathit{MS\_Sum})$$

The CHC translations, the transformed CHCs, and the models computed by Eldarica for all verification problems considered in this section and in the previous one are presented in the Appendix.

We think that our method based on the introduction of difference predicates works for many other verification problems with a structure similar to the ones presented in this paper. We leave for future work the task of making a thorough experimental evaluation.

4 Concluding Remarks

Let us briefly discuss how the transformation-based verification technique presented in this paper, through some sorting examples, can be applied to a larger class of verification problems. We will consider in particular: (A) the correctness issue of that technique, and (B) the mechanization of that technique.

Point (A). The main hypothesis needed to show correctness of our technique is that the predicates occurring in the initial set of clauses define total functional relations. This property is guaranteed by construction in the case when those predicates are the CHC translation of functional programs that terminate for all inputs. One more hypothesis that we need is the functionality of the difference predicates introduced during the transformation process. This functionality requirement can be checked in the model computed by the CHC solver, which is expressed as a set of LIA and/or Bool constraints. Note, however, that even if functionality of the difference predicates does not hold, it is the case that the satisfiability of the derived clauses implies the satisfiability of the initial clauses, and in many practical verification problems this implication is all that is required.

Point (B). In order to mechanize our transformation technique, we need to extend the Elimination Algorithm [8] with a suitable automated mechanism for introducing difference predicates. This mechanism can be based on the six steps shown in our examples of Sections 2 and 3. Among these, Step 3 (Matching) seems to be the most critical one, as many different matching substitutions could be computed for the same pair of clauses. However, once a matching has been chosen among the possible ones, the subsequent steps, and in particular the introduction of the difference predicates (Step 4), is straightforward. More sophisticated mechanisms may take into account the constraints occurring in the clauses, and may apply widening techniques which have been considered in other transformation methods [3, 15]. At the time of writing we have made some initial steps towards an implementation of the extended Elimination Algorithm that introduces difference predicates by using the VeriMAP transformation and verification system [4].

Various techniques for mechanizing induction over data structures have been proposed in the literature (see, for instance, the *rippling* technique [2]), but in the two-phase approach to program verification we apply in this paper we need not to take advantage of them. Indeed, in the first phase of our approach we transform using fold/unfold rules a set of clauses into a new, equisatisfiable set of clauses where the data structures have been eliminated, and thus no induction over them is required. Then, in the second phase of our approach, we do not need induction either, because satisfiability of the new set of clauses is shown by the use of CHC solvers acting over Linear Integer Arithmetic and Boolean constraints only.

To summarize, this paper presents ongoing work which follows a very general approach to program verification based on constrained Horn clauses. As also shown in the examples we have presented, the reduction of a program verification problem to a CHC satisfiability problem can often be obtained by a straightforward translation. However, proving the satisfiability of the clauses obtained by that translation is, in many cases, a much harder task. In a series of papers [3, 5, 6, 8, 7, 15] it has been shown that by combining various transformation techniques, such as *Specialization* and *Predicate Pairing*, we can derive equisatisfiable sets of clauses for which the efficacy of the CHC solvers is significantly improved. This approach avoids the burden of implementing very sophisticated solving strategies depending on the class of satisfiability problems one is required to solved. In particular, in the class of problems considered in this paper, consisting in checking the satisfiability of clauses over inductively defined data structures, we can avoid to implement *ad hoc* strategies to deal with induction proofs. In this paper we have also investigated the use of a novel transformation technique [9] based on the introduction of the so-called difference predicates and their relation with the lemmata that are often necessary in the inductive proofs of the properties of interest.

To the best of our knowledge, the only other approach that makes use of transformations of CHCs to remove inductively defined data structures is the one based on the notion of *CHC product* [18]. A CHC product is an operation that composes pairs of sets of clauses that have a similar recursive structure and, in some cases, it derives more compact sets of clauses with respect to those obtained by fold/unfold transformations. On the other hand, however, fold/unfold transformations are more general than CHC products, and often allow easier proofs of correctness. Moreover, no mechanism for lemma generation is provided when computing CHC products.

We leave it for future work to experiment on significant benchmarks and test whether our approach pays off in practice.

5 Acknowledgments

This work was supported by Gruppo Nazionale per il Calcolo Scientifico (GNCS-INdAM).
Emanuele De Angelis, Fabio Fioravanti, and Alberto Pettorossi are research associates at CNR-IASI, Rome, Italy.

References

[1] 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 inComputer Science 9300, Springer, pp. 24–51, doi:10.1007/978-3-319-23534-9_2.

[2] 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.

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

[4] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2014): *VeriMAP: A Tool for Verifying Programs through Transformations*. In: *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '14*, Lecture Notes in Com-

puter Science 8413, Springer, pp. 568–574, doi:10.1007/978-3-642-54862-8_47. Available at: http://www.map.uniroma2.it/VeriMAP.

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

[6] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2017): *Semantics-based generation of verification conditions via program specialization*. Science of Computer Programming 147, pp. 78–108, doi:10.1016/j.scico.2016.11.002. Available at http://www.sciencedirect.com/science/article/pii/S016764231630199X. Selected and Extended papers from the International Symposium on Principles and Practice of Declarative Programming 2015.

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

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

[9] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2019): *Lemma Generation for Horn Clause Satisfiability: A Preliminary Study*. In A. Lisitsa & A. Nemytykh, editors: Proceedings of VPT 2019 Workshop on Program Verification and Program Transformation, Genova, Italy, April 4th, 2019, To appear in EPTCS. Available at http://refal.botik.ru/vpt/vpt2019/VPT2019_paper_9.pdf.

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

[11] G. Fedyukovich, S. Prabhu, K. Madhukar & A. Gupta (2018): *Solving Constrained Horn Clauses Using Syntax and Data*. 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. 1–9, doi:10.23919/FMCAD.2018.8603011.

[12] 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.

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

[14] H. Hojjat & P. 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.

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

[16] A. Komuravelli, A. Gurfinkel & S. Chaki (2014): *SMT-Based Model Checking for Recursive Programs*. In A. Biere & R. Bloem, editors: Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, Lecture Notes in Computer Science 8559, Springer, pp. 17–34, doi:10.1007/978-3-319-08867-9_2.

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

[18] 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 Rea-

soning, Maun, Botswana, May 7-12, 2017, *EPiC Series in Computing* 46, EasyChair, pp. 338–355, doi:10.29007/gr5c. Available at https://easychair.org/publications/paper/LlxW.

[19] 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.

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

[21] 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.

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

6 Appendix

In this Appendix we present the proofs of some properties of various functional programs implementing sorting algorithms, including those presented in the previous sections. For every proof relative to a ⟨program, property⟩ pair we present:
(1) the set of constrained Horn clauses produced by the translation of the functional sorting program and the property to be proved,
(2) the set of constrained Horn clauses derived after the transformation based on the Elimination Algorithm [8] extended by the technique for introducing difference predicates presented in this paper, and
(3) the model of the constrained Horn clauses derived at Point (2) which has been computed by the CHC solver Eldarica.

Eldarica has been called by the command '`$eldarica -horn -hsmt -sol` *filename*', where *filename* is the file containing the SMT-LIB translation of the CHCs for which a model is sought.

Note that Eldarica has not been able to construct a model for any of the sets of constrained Horn clauses of Point (1) (that is, before the application of the Elimination Algorithm extended by the introduction of difference predicates) within the timeout of 120 second. All proofs have been conducted on a personal computer Intel Core i7-8550U CPU, 1.80GHz X 4 cores (8 threads) with 16 GB of RAM.

The ⟨program, property⟩ pairs we have considered in this appendix are:

1. ⟨*InsertionSort*, *IS_Perm*⟩
2. ⟨*InsertionSort*, *IS_Orderedness*⟩
3. ⟨*InsertionSort*, *IS_Length*⟩
4. ⟨*InsertionSort*, *IS_Sum*⟩
5. ⟨*SelectionSort*, *SS_Perm*⟩
6. ⟨*SelectionSort*, *SS_Orderedness*⟩
7. ⟨*SelectionSort*, *SS_Length*⟩
8. ⟨*QuickSort*, *QS_Perm*⟩
9. ⟨*QuickSort*, *QS_Sum*⟩
10. ⟨*MergeSort*, *MS_Sum*⟩

6.1 Program *InsertionSort* and property *IS_Perm*.

6.1.1 Constrained Horn clauses obtained by translating *InsertionSort* and *IS_Perm*.

```
:- pred ins(int,list(int),list(int)).
:- pred insertionSort(list(int),list(int)).
:- pred count(int,list(int),int).

false :- C1=\=C2, insertionSort(L,S), count(X,L,C1), count(X,S,C2).
ins(I,[],[I]).
ins(I,[X|Xs],[I,X|Xs]) :-  I=<X.
ins(I,[X|Xs],[X|Ys]) :-  I>X, ins(I,Xs,Ys).
insertionSort([],[]).
insertionSort([X|Xs],S) :- insertionSort(Xs,S1), ins(X,S1,S).
count(X,[],0).
count(X,[H|T],N) :- X=H, N=M+1, count(X,T,M).
count(X,[H|T],N) :- X=\=H, count(X,T,N).
```

6.1.2 Transformed constrained Horn clauses.

```
:- pred new1(int,int,int).
:- pred new2(int,int).
```

```
:- pred new3(int,int,int).
:- pred new4(int,int,int,int).
:- pred new5(int,int).
:- pred new6(int,int,int,int).
:- pred diff1(int,int,int).
:- pred diff2(int,int,int,int).

false :- N1=\=N2, new1(X,N1,N2).
new1(X,0,0).
new1(X,N1,N2) :- N1=N+1, new1(X,N,N2a), diff1(X,N2a,N2).
new1(X,N1,N2) :- X=\=Y, new1(Y,N1,N2b), diff2(X,Y,N2b,N2).
diff1(X,0,N2)  :- N2=N1+1, new2(X,N1).
diff1(X,N1,N2) :- N2=M2+1, N1=M1+1, new3(X,M2,M1).
diff1(X,N1,N2) :- X=<Y, N2=N+1, X=\=Y, new4(X,Y,N,N1).
diff2(X,Y,0,0) :- Y=\=X.
diff2(X,Y,M,N) :- X=<Y, Y=\=X, M=K+1, new3(Y,N,K).
diff2(X,Y,M,N) :- X=<Z, Y=\=X, Y=\=Z, N=M, new5(Y,N).
diff2(X,Y,M,N) :- X>Y, N=H+1, M=K+1, diff2(X,Y,K,H).
new2(X,0).
new3(X,N1,N)  :- N1=N+1, new5(X,N).
new4(X,Y,N,N) :- X=<Y, X=\=Y, new5(X,N).
new5(X,0).
new5(X,N1) :- N1=N+1, new5(X,N).
```

6.1.3 Model of the transformed constrained Horn clauses.

```
diff1(A,B,C) :- (((B - C) = -1), (B >= 0)).
diff2(A,B,C,D) :- ((C = D), (C >= 0)).
new1(A,B,C) :- ((B = C), (B >= 0)).
new2(A,B) :- (B = 0).
new3(A,B,C) :- (((C - B) = -1), (B >= 1)).
new4(A,B,C,D) :- ((D = C), ((C >= 0), ((B - A) >= 1))).
new5(A,B) :- (B >= 0).
```

6.2 Program *InsertionSort* and property *IS_Orderedness*.

6.2.1 Constrained Horn clauses obtained by translating *InsertionSort* and *IS_Orderedness*.

```
:- pred ins(int,list(int),list(int)).
:- pred insertionSort(list(int),list(int)).
:- pred ordered(list(int),bool).

false :- insertionSort(L,S), ordered(S,false).
ins(I,[],[I]).
ins(I,[X|Xs],[I, X| Xs]) :- I=<X.
ins(I,[X|Xs],[X|Ys]) :- I>X, ins(I,Xs,Ys).
insertionSort([],[]).
```

```
insertionSort([X|Xs],S) :- insertionSort(Xs,S1), ins(X,S1,S).
ordered([],true).
ordered([X],true).
ordered([X,Y|T],false) :-X>Y.
ordered([X,Y|T],B) :-X=<Y, ordered([Y|T],B).
```

6.2.2 Transformed constrained Horn clauses.

```
:- pred diff(int,bool,bool).
:- pred new1(bool).
:- pred new2(int,int,int,bool,bool).

false :- new1(false).
new1(true).
new1(B) :- new1(B1), diff(X,B,B1).
diff(I,true,true).
diff(I,B,B).
diff(I,false,false).
diff(I,B,B1) :- new2(I,Y1,Y,B,B1).
new2(I,Y1,Y,B,B) :- I=<Y1, I=Y.
new2(I,Y1,Y,false,false) :- I>Y, Y1=Y.
new2(I,Y1,Y,true,true) :- I>Y, Y1=Y.
new2(I,Y1,Y,B,B1) :- I>Y, Y=<Y2, Y=<Y3, Y1=Y, new2(I,Y3,Y2,B,B1).
```

6.2.3 Model of the transformed constrained Horn clauses.

```
new1(A) :- (A = true).
diff(A,B,C) :- (\+((C = true)); (B = true)).
new2(A,B,C,D,E) :- (\+((E = true)); (D = true)).
```

6.3 Program *InsertionSort* and property *IS_Length*.

6.3.1 Constrained Horn clauses obtained by translating *InsertionSort* and *IS_Length*.

```
:- pred length(list(int),int).
:- pred delete(int,list(int),list(int)).
:- pred ins(int,list(int),list(int)).
:- pred insertionSort(list(int),list(int)).

false :- N1>=0, N2>=0, N1 =\= N2,
    length(L,N1), insertionSort(L,S), length(S,N2).
length([],0).
length([X|Xs],N) :- N1>=0, N = N1+1, length(Xs,N1).
delete(X,[Y|T],T) :- X=Y.
delete(X,[Y|T],[Y|D]) :- X=\=Y, delete(X,T,D).
ins(I,[],[I]).
ins(I,[X|Xs],[I, X| Xs]) :- I=<X-1.
```

```
ins(I,[X|Xs],[X|Ys]) :- I>=X, ins(I,Xs,Ys).
insertionSort([],[]).
insertionSort([X|Xs],S) :- ins(X,S1,S), insertionSort(Xs,S1).
```

6.3.2 Transformed constrained Horn clauses.

```
:- pred diff(int,int,int).
:- pred new1(int,int).
:- pred new2(int).

false :- new1(N1,N2), N1 >= 0, N2 >= 0, N1 =\= N2.
new1(0,0).
new1(N1,N2) :- N11>=0, N1=N11+1, new1(N11,N3), diff(X,N2,N3).
diff(I,N2,N3) :- N2=1, N3=0.
diff(I,N2,N3) :- I=<X-1, N2=N3+1, N3=N5+1, new2(N5).
diff(I,N2,N3) :- I>=X, N2=N4+1, N3=N5+1, diff(I,N4,N5).
new2(0).
new2(N) :- N1=N+1, new2(N1).
```

6.3.3 Model of the transformed constrained Horn clauses.

```
diff(A,B,C) :- ((B - C) = 1).
new1(A,B) :- ((A = B), (A >= 0)).
new2(A) :- (A =< 0).
```

6.4 Program *InsertionSort* and property *IS_Sum*.

6.4.1 Constrained Horn clauses obtained by translating *InsertionSort* and *IS_Sum*.

```
:- pred sumlist(list(int),int).
:- pred ins(int,list(int),list(int)).
:- pred insertionSort(list(int),list(int)).

false :- M=\=N, sumlist(L,M), insertionSort(L,S), sumlist(S,N).
sumlist([],0).
sumlist([X|Xs],M) :- M=X+N, sumlist(Xs,N).
ins(I,[],[I]).
ins(I,[X|Xs],[I,X|Xs]) :- I=<X.
ins(I,[X|Xs],[X|Ys]) :- I>X, ins(I,Xs,Ys).
insertionSort([],[]).
insertionSort([X|Xs],S) :- insertionSort(Xs,SXs), ins(X,SXs,S).
```

6.4.2 Transformed constrained Horn clauses.

```
:- pred diff(int,int,int).
:- pred new1(int,int).
:- pred new2(int).
```

```
false :- M=\=N, new1(M,N).
new1(0,0).
new1(M1,N1) :- M1=H+M0, new1(M0,N0), diff(H,N0,N1).
diff(H,0,N1)  :- N1=H.
diff(H,N0,N1) :- H=<X, N0=X+N2, N1=H+N0, new2(N2).
diff(H,N0,N1) :- H>X,  N0=X+N2, N1=X+N3, diff(H,N2,N3).
new2(0).
new2(N) :- N=X+N1, new2(N1).
```

6.4.3 Model of the transformed constrained Horn clauses.

```
diff(A,B,C) :- ((A + (B - C)) = 0).
new1(A,B) :- (A = B).
new2(A) :- true.
```

6.5 Program *SelectionSort* and property *SS_Perm*.

6.5.1 Constrained Horn clauses obtained by translating *SelectionSort* and *SS_Perm*.

```
:- pred min(list(int),int).
:- pred delete(int,list(int),list(int)).
:- pred selectionSort(list(int),list(int)).
:- pred count(int,list(int),int).

false :- N1 =\= N2, count(X,L,N1), selectionSort(L,S), count(X,S,N2).
min([X],X).
min([X|T],M) :- X=<M1, M=X, min(T,M1).
min([X|T],M) :- X>M1, M=M1, min(T,M1).
delete(X,[Y|T],T) :- X=Y.
delete(X,[Y|T],[Y|D]) :- X=\=Y, delete(X,T,D).
selectionSort([],[]).
selectionSort(L,[M|T]) :- min(L,M), delete(M,L,L1), selectionSort(L1,T).
count(X,[],0).
count(X,[H|T],N) :- X=H, N=M+1, count(X,T,M).
count(X,[H|T],N) :- X=\=H, count(X,T,N).
```

6.5.2 Transformed constrained Horn clauses.

```
:- pred new1(int,int,int).
:- pred diff1(int,int,int,int).
:- pred diff2(int,int,int,int).
:- pred new2(int,int,int).

false :- N1 =\= N2, new1(X,N1,N2).
new1(A,0,0).
new1(A,B,C) :- C=H+1, A=E, diff1(A,E,B,B1), new1(A,B1,H).
```

```
new1(A,B,C) :- A=\=E, diff2(A,E,B,B2), new1(A,B2,C).
diff1(A,A,1,0).
diff1(A,A,B1,B) :- B1=B+1, A=<C, new2(A,B,C).
diff1(A,A,B,C) :- D>E, A=E, diff1(A,E,B,C).
diff2(A,B,C1,D1) :- A>B, C1=C+1, D1=D+1, diff2(A,B,C,D).
diff2(A,B,0,0) :- A=\=B.
diff2(A,B,C,C) :- A=\=B, B=<D, new2(A,C,D).
diff2(A,B,C,D) :- A=\=B, A=\=E, E>B, diff2(A,B,C,D).
new2(A,B,C) :- A=C, B=1.
new2(A,B1,A) :- A=<C, B1=B+1, new2(A,B,C).
new2(A,B1,C) :- A>C, B1=B+1, new2(A,B,C).
new2(A,0,B) :- A=\=B.
new2(A,B,C) :- A=\=C, C=<E, new2(A,B,E).
new2(A,B,C) :- A=\=D, D>C, new2(A,B,C).
```

6.5.3 Model of the transformed constrained Horn clauses.

```
diff1(A,B,C,D) :- (((C - D) = 1), (C >= 1)).
diff2(A,B,C,D) :- ((C = D), (C >= 0)).
new1(A,B,C) :- ((B = C), (B >= 0)).
new2(A,B,C) :- (B >= 0).
```

6.6 Program *SelectionSort* and property *SS_Orderedness.*

6.6.1 Constrained Horn clauses obtained by translating *SelectionSort* and *SS_Orderedness.*

```
:- pred min(list(int),int).
:- pred delete(int,list(int),list(int)).
:- pred selectionSort(list(int),list(int)).
:- pred ordered(list(int),bool).

false :- selectionSort(L,S), ordered(S,false).
min([X],X).
min([X|T],M) :- X=<M1, M=X, min(T,M1).
min([X|T],M) :- X>M1, M=M1, min(T,M1).
delete(X,[Y|T],T) :- X=Y.
delete(X,[Y|T],[Y|D]) :- X=\=Y, delete(X,T,D).
selectionSort([],[]).
selectionSort(L,[M|T]) :- min(L,M), delete(M,L,L1), selectionSort(L1,T).
ordered([],true).
ordered([X],true).
ordered([X,Y|T],false) :- X>Y.
ordered([X,Y|T],B) :- X=<Y, ordered([Y|T],B).
```

6.6.2 Transformed constrained Horn clauses.

```
:- pred new1(bool).
```

```
:- pred new2(int,int).

false :- new1(false).
new1(true).
new1(false) :- new2(M,Y), M>Y.
new1(B) :- new1(B).
new2(M,Y) :- X>M, new2(M,Y), M>Y.
```

6.6.3 Model of the transformed constrained Horn clauses.

```
new1(A) :- (A = true).
new2(A,B) :- false.
```

6.7 Program *SelectionSort* and property *SS_Length*.

6.7.1 Constrained Horn clauses obtained by translating *SelectionSort* and *SS_Length*.

```
:- pred length(list(int),int).
:- pred min(list(int),int).
:- pred delete(int,list(int),list(int)).
:- pred selectionSort(list(int),list(int)).

false :- N1>=0, N2>=0, N1=\=N2,
    length(L,N1), selectionSort(L,S), length(S,N2).
length([],0).
length([X|Xs],N) :- N1>=0, N=N1+1, length(Xs,N1).
min([X],X).
min([X|T],M) :- X=<M1, M=X, min(T,M1).
min([X|T],M) :- X>M1, M=M1, min(T,M1).
delete(X,[Y|T],T) :- X=Y.
delete(X,[Y|T],[Y|D]) :- X=\=Y, delete(X,T,D).
selectionSort([],[]).
selectionSort(L,[M|T]) :- min(L,M), delete(M,L,L1), selectionSort(L1,T).
```

6.7.2 Transformed constrained Horn clauses.

```
:- pred new1(int,int).
:- pred diff(int,int,int).
:- pred new2(int,int).

false :- N1>=0, N2>=0, N1=\=N2, new1(N1,N2).
new1(0,0).
new1(N1,N2) :- N3>=0, N2=N3+1, new1(N4,N3), diff(X,N1,N4).
diff(H,1,0).
diff(X,N1,N2) :- N2>=0, N1=N2+1, X=<M1, new2(N2,M1).
diff(X,N1,N4) :- N2>=0, N1=N2+1, H>X, N5>=0, N4=N5+1, diff(X,N2,N5).
new2(1,H).
```

```
new2(N2,H)  :- N2=N5+1, H=<M2, new2(N5,M2).
new2(N2,M1) :- N2=N5+1, H>M1, new2(N5,M1).
```

6.7.3 Model of the transformed constrained Horn clauses.

```
diff(A,B,C) :- (((((B - C) = 1), (B >= 3)); ((B = 2), (C = 1)));
    ((B = 1), (C = 0))).
new1(A,B) :- ((A = B), (A >= 0)).
new2(A,B) :- (A >= 1).
```

6.8 Program *QuickSort* and property *QS_Perm*.

6.8.1 Constrained Horn clauses obtained by translating *QuickSort* and *QS_Perm*.

```
:- pred append(list(int),list(int),list(int)).
:- pred partition(int,list(int),list(int),list(int)).
:- pred quickSort(list(int),list(int)).
:- pred count(int,list(int),int).

false :- N1 =\= N2, count(X,L,N1), quickSort(L,S), count(X,S,N2).
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
partition(X,[],[],[]).
partition(X,[Y|Ys],[Y|L1],L2) :- Y=<X, partition(X,Ys,L1,L2).
partition(X,[Y|Ys],L1,[Y|L2]) :- X<Y, partition(X,Ys,L1,L2).
quickSort([],[]).
quickSort([X|Xs],S) :- partition(X,Xs,Ys,Zs),
   quickSort(Ys,S1), quickSort(Zs,S2), append(S1,[X|S2],S).
count(X,[],0).
count(X,[H|T],N) :- X=H, N=M+1, count(X,T,M).
count(X,[H|T],N) :- X=\=H, count(X,T,N).
```

6.8.2 Transformed constrained Horn clauses.

```
:- pred new1(int,int,int).
:- pred new2(int,int).
:- pred new3(int,int,int,int,int).
:- pred new4(int,int,int,int,int).
:- pred diff1(int,int,int,int,int).
:- pred diff2(int,int,int,int,int).
:- pred diff3(int,int,int,int,int).
:- pred diff4(int,int,int,int,int).

false :- M=\=N, new1(X,M,N).
new1(X,0,0).
new1(X,M,N) :- M=K+1, X=Y, diff1(X,Y,K,Ma,Mb), new1(X,Ma,Na),
    new1(X,Mb,Nb), diff2(X,Y,N,Na,Nb).
```

```
new1(X,M,N) :- X=\=Y, diff3(X,Y,M,Ma,Mb), new1(X,Ma,Na),
    new1(X,Mb,Nb), diff4(X,Y,N,Na,Nb).
diff1(X,Y,0,0,0)   :- X=Y.
diff1(X,Y,M,Ma,Mb) :- X=Y, Ma=Ma1+1, M=M1+1, diff1(X,Y,M1,Ma1,Mb).
diff2(X,Y,N,0,Nb)  :- X=Y, N=Nb+1, new2(X,Nb).
diff2(X,Y,N,Na,Nb) :- X=Y, Na=Na1+1, N=N1+1, diff2(X,Y,N1,Na1,Nb).
diff3(X,Y,M,Ma,Mb) :- X<Y, new3(X,Y,M,Ma,Mb).
diff3(X,Y,M,Ma,Mb) :- Y<X, new4(X,Y,M,Ma,Mb).
diff4(X,Y,N,0,N)   :- X=\=Y, new2(X,N).
diff4(X,Y,N,Na,Nb) :- X=\=Y, Na=Na1+1, N=N1+1, diff4(X,Y,N1,Na1,Nb).
new2(X,0).
new2(X,N) :- N=N1+1, new2(X,N1).
new3(X,Y,0,0,0)   :- X<Y.
new3(X,Y,M,Ma,Mb) :- X<Y, M=M1+1, Ma=Ma1+1, new3(X,Y,M1,Ma1,Mb).
new4(X,Y,0,0,0)   :- X>Y.
new4(X,Y,M,Ma,Mb) :- Y<X, M=M1+1, Mb=Mb1+1, new4(X,Y,M1,Ma,Mb1).
```

6.8.3 Model of the transformed constrained Horn clauses.

```
diff1(A,B,C,D,E) :- (((((((A = B), (C = 2)), (D = 2)), (E = 0));
  ((((A = B), (C = 1)), (D = 1)), (E = 0)));
  (((C = D), (E = 0)), ((((B + C) - A) >= 3), ((A + (C - B)) >= 3))));
  (((C = 0), (D = 0)), (E = 0))).
diff2(A,B,C,D,E) :- (((
  (((C - D) = 1), (((((B + C) - A) >= 2), ((A + (C - B)) >= 2)), (E >= 0)));
  ((((C - E) = 1), (D = 0)), ((((B + C) - A) >= 2), ((A + (C - B)) >= 2))));
  (((C = 1), (D = 0)), (E = 0)));
  ((((((B + C) - A) >= 2),
    (((B + D) - A) >= 1)), ((A + (C - B)) >= 2)), ((A + (D - B)) >= 1)),
    (E >= 1))).
diff3(A,B,C,D,E) :- (((((((C = D), (E = 0)), (C >= 3));
  (((C = E), (D = 0)), (C >= 2))); (((C = 2), (D = 2)),
  (E = 0))); (((C = 1), (D = 1)), (E = 0))); (((C = 1), (D = 0)), (E = 1)));
  (((C = 0), (D = 0)), (E = 0))).
diff4(A,B,C,D,E) :- (((((((C = D), (E = 0)), (C >= 1));
  (((C = E), (D = 0)), (C >= 2)));
  (((C = 1), (D = 0)), (E = 1))); (((C = 0), (D = 0)), (E = 0)));
  ((E = 1), ((C >= 0), (D >= 1))));
  (((C >= 0), (D >= 1)), (E >= 2))).
new1(A,B,C) :- ((((B = C), (B >= 2)); ((B = 1), (C = 1))); ((B = 0), (C = 0))).
new2(A,B) :- (B >= 0).
new3(A,B,C,D,E) :- ((((((C = D), (E = 0)), (C >= 3));
  (((C = 2), (D = 2)), (E = 0)));
  (((C = 1), (D = 1)), (E = 0))); (((C = 0), (D = 0)), (E = 0))).
new4(A,B,C,D,E) :- (((((C = E), (D = 0)), (C >= 2));
  (((C = 1), (D = 0)), (E = 1))); (((C = 0), (D = 0)), (E = 0))).
```

6.9 Program *QuickSort* and property *QS_Sum*.

6.9.1 Constrained Horn clauses obtained by translating *QuickSort* and *QS_Sum*.

```
:- pred sumlist(list(int),int).
:- pred quickSort(list(int),list(int)).
:- pred partition(int,list(int),list(int),list(int)).
:- pred append(list(int),list(int),list(int)).

false :- N1 =\= N2, sumlist(L,N1), quickSort(L,S), sumlist(S,N2).
sumlist([],0).
sumlist([X|Xs],S) :- S=S1+X, sumlist(Xs,S1).
quickSort([],[]).
quickSort([D|T],S) :- partition(D,T,T1,T2), quickSort(T1,S1),
    quickSort(T2,S2), append(S1,[D|S2],S).
partition(D,[],[],[]).
partition(D,[H|T],[H|L1],L2) :- H=<D,   partition(D,T,L1,L2).
partition(D,[H|T],L1,[H|L2]) :- D=<H-1, partition(D,T,L1,L2).
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

6.9.2 Transformed constrained Horn clauses.

```
:- pred new1(int,int).
:- pred diff1(int,int,int,int).
:- pred diff2(int,int,int,int).

false :- new1(N1,N2), N1 =\= N2.
new1(0,0).
new1(N1,N2) :- N1=N3+H, diff1(N3,H,K1,K2), new1(K1,J1), new1(K2,J2),
    diff2(H,N2,J1,J2).
diff1(0,H,0,0).
diff1(N3,H,K1,K2) :- N3=N4+X, H>=X, K1=K3+X, diff1(N4,H,K3,K2).
diff1(N3,H,K1,K2) :- N3=N4+X, H=<X-1, K2=K4+X, diff1(N4,H,K1,K4).
diff2(H,N2,0,N3)  :- N2=H+N3.
diff2(H,N2,J1,J2) :- N2=N3+Y, J1=J3+Y, diff2(H,N3,J3,J2).
```

6.9.3 Model of the transformed constrained Horn clauses.

```
diff1(A,B,C,D) :- ((A + ((-1 * C) - D)) = 0).
diff2(A,B,C,D) :- ((A + ((C + D) - B)) = 0).
new1(A,B) :- (A = B).
```

6.10 Program *MergeSort* and property *MS_Sum*.

6.10.1 Constrained Horn clauses obtained by translating *MergeSort* and *MS_Sum*.

```
:- pred sumlist(list(int),int).
```

```
:- pred merge(list(int),list(int),list(int)).
:- pred mergeSort(list(int),list(int)).
:- pred split(list(int),list(int),list(int)).
:- pred mydiv(int,int,int).
:- pred length(list(int),int).
:- pred takedrop(int,list(int),list(int),list(int)).

false :- N1 =\= N2, sumlist(L,N1), mergeSort(L,S), sumlist(S,N2).
sumlist([],0).
sumlist([X|Xs],S) :- S=S1+X, sumlist(Xs,S1).
mergeSort([],[]).
mergeSort([H],[H]).
mergeSort([H|T],S) :- T=[Y|T1], split([H|T],L1,L2),
    mergeSort(L1,S1), mergeSort(L2,S2), merge(S1,S2,S).

split(L,L1,L2) :- length(L,N), mydiv(N,2,N1), takedrop(N1,L,L1,L2).
mydiv(N,2,N1) :- N=N1+N2, N1-N2=<1.
length([],0).
length([H|T],N) :- N=M+1, length(T,M).
takedrop(N,[],[],[]).
takedrop(0,L,[],L).
takedrop(N,[H|T],[H|T1],L) :- N>=1, N1=N-1, takedrop(N1,T,T1,L).
merge([],L,L).
merge([X|Xs],[],[X|Xs]).
merge([X|Xs],[Y|Ys],[X|Zs]) :- X=<Y, merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[Y|Ys],[Y|Zs]) :- X>=Y+1, merge([X|Xs],Ys,Zs).
```

6.10.2 Transformed constrained Horn clauses.

```
:- pred new1(int,int).
:- pred diff1(int,int,int,int,int).
:- pred diff2(int,int,int).
:- pred new2(int,int).
:- pred new3(int,int,int,int,int).
:- pred new4(int).
:- pred new5(int,int,int,int).
:- pred new6(int,int,int,int).

false :- new1(N1,N2), N1 =\= N2.
new1(0,0).
new1(H,H).
new1(N1,N2) :- N1=N3+X, diff1(Y,N3,X,K1,K2), new1(K1,J1),
    new1(K2,J2), diff2(N2,J1,J2).
diff1(A,B,C,0,D) :- B=E+A, H>=0, I=H+1, G>=0, H=G+1, I=0+J,
    0-J=<1, D=K+C, K=E+A, new2(E,G).
diff1(A,B,C,D,E) :- B=F+A, I>=0, J=I+1, H>=0, I=H+1, J=K+L,
    K-L=<1, K>=1, 0=K-1, D=0+C, E=F+A, new2(F,H).
```

```
diff1(A,B,C,D,E) :- B=F+A, J>=0, K=J+1, G>=0, J=G+1,
    K=L+M, L-M=<1, L>=1, N=L-1, N>=1, H=N-1, D=O+C, O=I+A.
diff2(A,0,B) :- new4(A), A=B, new3(F,G,H,I,E).
diff2(A,B,0) :- A=C+D, A=B, new4(C).
diff2(A,B,C) :- D=<E, A=F+D, B=G+D, C=H+E, new5(E,F,G,H).
diff2(A,B,C) :- D>=E+1, A=F+E, B=G+D, C=H+E, new6(D,F,G,H).
new2(0,0).
new2(C,D) :- C=E+A, F>=0, D=F+1, new2(E,F).
new3(0,0,A,0,0) :- A>=0.
new3(0,0,0,0,0) :- 0>=0.
new3(A,B,0,0,C) :- D>=0, B=D+1, A=F+G, C=F+G, new2(F,D).
new3(A,B,C,D,E) :- C>=0, F>=0, B=F+1, C>=1, H=C-1, H>=0,
    A=G+J, D=I+J, new3(G,F,H,I,E).
new4(0).
new4(A) :- A=B+C, new4(B).
new5(A,B,0,C) :- B=C+A, new4(C).
new5(A,B,C,D) :- E=<A, B=F+E, C=G+E, new5(A,F,G,D).
new5(A,B,C,D) :- E>=A+1, B=F+A, C=G+E, new6(E,F,G,D).
new6(A,B,C,0) :- B=C+A, new4(C).
new6(A,B,C,D) :- A=<E, B=F+A, D=G+E, new5(E,F,C,G).
new6(A,B,C,D) :- A>=E+1, B=F+E, D=G+E, new6(A,F,C,G).
```

6.10.3 Model of the transformed constrained Horn clauses.

```
diff1(A,B,C,D,E) :- ((B + (C + ((-1 * D) - E))) = 0).
diff2(A,B,C) :- ((A + ((-1 * B) - C)) = 0).
new1(A,B) :- (A = B).
new2(A,B) :- (B >= 0).
new3(A,B,C,D,E) :- (((A + ((-1 * D) - E)) = 0), ((B >= 0), (C >= 0))).
new4(A) :- true.
new5(A,B,C,D) :- ((A + ((C + D) - B)) = 0).
new6(A,B,C,D) :- ((A + ((C + D) - B)) = 0).
```