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

Emanuele De Angelis, Univ. Chieti-Pescara, Italy

Fabio Fioravanti, Univ. Chieti-Pescara, Italy

Alberto Pettorossi, Univ. Roma Tor Vergata, Italy

Maurizio Proietti, CNR-IASI, Italy

Overview

- Problem: Verifying properties of *functional programs on recursive datatypes*;
- Translating program properties into *Constrained Horn Clauses* on recursive datatypes;
- *Transforming* CHCs on recursive datatypes into equisatisfiable CHCs *on integers and booleans*;
- Verification of *linear recursive sorting* programs by introduction of *difference predicates*;
- Verification of *non-linear recursive sorting* programs.

Functional programs on recursive datatypes

- Statically typed, call-by-value, first order, functional language (OCaml).
- Computing the sum and the maximum of the absolute values of the elements of a list:

```
type list = Nil | Cons of int * list;;

let rec listsum l = match l with
  | Nil -> 0
  | Cons(x, xs) → (abs x) + listsum xs;;

let rec listmax l = match l with
  | Nil -> 0
  | Cons(x, xs) → let m = listmax xs in max (abs x) m;;

let main l = assert(listsum l >= listmax l);;
```

% Property:
% ∀l. listsum l >= listmax l

Translation into CHCs

- The functional program and property are translated into CHCs:

```
false :- S<M, listsum(L,S), listmax(L,M).                    % Query
max(X,Y,Z) :- (X=<Y-1, Z=Y) ; (X>=Y, Z=X).
abs(X,A) :- (X>=0, A=X); (X=<0, A=-X).
listsum([],S) :- S=0.
listsum([X|Xs],S) :- S=S1+A, abs(X,A), listsum(Xs,S1).
listmax([],M) ← M=0.
listmax([X|Xs],M) ← abs(X,A), max(A,M1,M), listmax(Xs,M1).
```

- The property holds iff the clauses are satisfiable;
  Indeed clauses are satisfiable but models are infinite disjunctions:
  - listsum(L,S) :- (L=[], S=0) ; (L=[A], abs(X,A)) ; …
  - listmax(L,M) :- (L=[], M=0) ; (L=[Az], abs(X,A)) ; …

  % Eldarica syntax
  % for models

- CHC solvers (Eldarica, Z3) over the quantifier-free Theory of Lists and Linear Integer Arithmetic (LIA) **cannot solve** them (i.e., construct a model).

Solving CHCs on recursive datatypes

- Approach 1: Extend CHC solving by *induction principles*: [Reynolds-Kuncak 2015, Unno-Torii-Sakamoto 2017].
- Approach 2 (this talk): Transform CHCs on inductive data types into equisatisfiable CHCs without recursive datatypes (e.g., on integers or booleans only). [Mordvinov-Fedyukovich 2017, DFPP 2018]
- Transformations inspired by techiques for eliminating inductive data structures: Deforestation [Wadler ‘88], Unnecessary Variable Elimination by Unfold/Fold [PP ‘91], Conjunctive Partial Deduction + Redundant Argument Filtering [DeSchreye et al. ‘99]

ADT Elimination Algorithm

- false :- S<M, listsum(L,S), listmax(L,M). % L existential list
- Define a new predicate:

  list-sum-max(S,M) :- listsum(L,S), listmax(L,M).
- Unfold:

  list-sum-max(S,M) :- S=0, M=0.

  list-sum-max(S,M) :- S=S1+A, abs(X,A), max(A,M1,M),
  listsum(Xs,S1), listmax(Xs,M1).

  Variant conjunctions
- Fold (eliminate lists):

  false :- S<M, **list-sum-max**(S,M).

  list-sum-max(S,M) :- S=0, M=0.

  list-sum-max(S,M) :- S=S1+A, abs(X,A), max(A,M1,M), % No lists
  list-sum-max(S1,M1).

Solving CHCs on LIA

```
false :- S<M, list-sum-max(S,M).
list-sum-max(S,M) :- S=0, M=0.
list-sum-max(S,M) :- S=S1+A, abs(X,A), max(A,M1,M),
                     list-sum-max(S1,M1).
```

- Equisatisfiability guaranteed by fold/unfold rules.
- No infinite models and are needed to show satisfiabilty.
- Solved by Eldarica (and Z3) without induction rules.
  LIA-definable model:

  list-sum-max(S,M) :- S>=M, M>=0.

Insertion Sort (Permutation)

```
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 iSort l = match l with
  | Nil -> Nil
  | Cons(x,xs) -> ins x (iSort 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;;

let main x l = assert(count x l = count x (iSort l));;
```

% Property: l and s have the same elements (counting duplicates).
% ∀l,s,x,n1,n2. (count x l = n1) ∧ (iSort l = s) ∧ (count x s = n2) → n1=n2

Insertion Sort: Translation into CHCs

```
false :- N1=\=N2, count(X,L,N1), iSort(L,S), count(X,S,N2).
ins(A,[ ],[A]).
ins(A,[X|Xs],[A,X|Xs]) :- A=<X.
ins(A,[X|Xs],[X|Ys]) :- A>X, ins(A,Xs,Ys).
iSort([ ],[ ]).
iSort([X|Xs],S) :- iSort(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).
```

- CHC solvers (Eldarica, Z3) over the quantifier-free theory of lists and Linear Integer Arithmetic (LIA) **cannot solve** these clauses.

Insertion Sort: Applying the Elimination Algorithm

```
false :- N1=\=N2, count(X,L,N1), iSort(L,S), count(X,S,N2).      % L,S existential lists
```

Insertion Sort: Applying the Elimination Algorithm

false :- N1=\=N2, count(X,L,N1), iSort(L,S), count(X,S,N2). % L,S existential lists

Define a new predicate (and Rename Variables):
***new*1**(X',N1',N2') :- count(X',L',N1'), iSort(L',S'), count(X',S',N2').

Insertion Sort: Applying the Elimination Algorithm

```
false :- N1=\=N2, count(X,L,N1), iSort(L,S), count(X,S,N2).
```

% **L**,**S** existential lists

Define a new predicate (and Rename Variables):

***new*1**(X’,N1’,N2’) :- count(X’,**L’**,N1’), iSort(**L’**,**S’**), count(X’,**S’**,N2’).

Unfold:

***new*1**(X,0,0).

***new*1**(X,N1,N2) :- N1=N+1, count(X,**Xs**,N), iSort(**Xs**,**S1**), ins(X,**S1**,**S**), count(X,**S**,N2).

***new*1**(X,N1,N2) :- X=\=Y, count(X,Xs,N1), iSort(Xs,S1), ins(Y,S1,S), count(X,S,N2).

Insertion Sort: Embed

false :- N1=\=N2, count(X,L,N1), iSort(L,S), count(X,S,N2).

% L,S existential lists

Define a new predicate (and Rename Variables):

***new*1**(X',N1',N2') :- count(X',L',N1'), iSort(L',S'), count(X',S',N2').

variant atoms

but NOT variant conjunction

Unfold:

***new*1**(X,0,0).

***new*1**(X,N1,N2) :- N1=N+1, count(X,Xs,N), iSort(Xs,S1), ins(X,S1,S), count(X,S,N2).

***new*1**(X,N1,N2) :- X=\=Y, count(X,Xs,N1), iSort(Xs,S1), ins(Y,S1,S), count(X,S,N2).

Insertion Sort: Embed

false :- N1=\=N2, count(X,L,N1), iSort(L,S), count(X,S,N2).

% L,S existential lists

Define a new predicate (and Rename Variables):

***new*1**(X',N1',N2') :- count(X',L',N1'), iSort(L',S'), count(X',S',N2').

variant atoms

but NOT variant conjunction

Unfold:

***new*1**(X,0,0).

***new*1**(X,N1,N2) :- N1=N+1, count(X,Xs,N), iSort(Xs,S1), ins(X,S1,S), count(X,S,N2).

***new*1**(X,N1,N2) :- X=\=Y, count(X,Xs,N1), iSort(Xs,S1), ins(Y,S1,S), count(X,S,N2).

Folding is not possible (The Elimination Algorithm does not terminate)

Insertion Sort: Embed

false :- N1=\=N2, count(X,L,N1), iSort(L,S), count(X,S,N2). % L,S existential lists

Define a new predicate (and Rename Variables):

***new*1**(X',N1',N2') :- count(X',L',N1'), iSort(L',S'), count(X',S',N2').

variant atoms
but NOT variant conjunction

Unfold:

***new*1**(X,0,0).

***new*1**(X,N1,N2) :- N1=N+1, count(X,Xs,N), iSort(Xs,S1), ins(X,S1,S), count(X,S,N2).

***new*1**(X,N1,N2) :- X=\=Y, count(X,Xs,N1), iSort(Xs,S1), ins(Y,S1,S), count(X,S,N2).

Folding is not possible (The Elimination Algorithm does not terminate)

New idea of this work: Match; Introduce **DIFFERENCE PREDICATES;** Replace; Fold

Insertion Sort: Match

Match Clause to be folded against definition

Define:
***new*1**(X',N1',N2') :- count(X',**L'**,N1'), iSort(**L'**,**S'**), count(X',**S'**,N2').

MATCH MISMATCH

Unfold: σ = {**Xs**/**L'**, **S1**/**S'**, X/X', N/N1'}
...
***new*1**(X,N1,N2) :- N1=N+1, count(X,**Xs**,N), iSort(**Xs**,**S1**), ins(X,**S1**,**S**), count(X,**S**,N2).

Insertion Sort: Match

Match Clause to be folded against definition

Define:

***new*1**(X',N1',N2') :- count(X',**L'**,N1'), iSort(**L'**,**S'**), count(X',**S'**,N2').

MATCH MISMATCH

Unfold: σ = {**Xs**/**L'**, **S1**/**S'**, X/X', N/N1'}

...

***new*1**(X,N1,N2) :- N1=N+1, count(X,**Xs**,N), iSort(**Xs**,**S1**), ins(X,**S1**,**S**), count(X,**S**,N2).

Apply substitution σ: σ

***new*1**(X',N1,N2) :- N1=N1'+1, count(X',**L'**,N1'), iSort(**L'**,**S'**), ins(X',**S'**,**S**), count(X',**S**,N2).

Insertion Sort: Difference Predicate

Introduce DIFFERENCE Predicate:

Difference between clause to be folded ("We have") and the definition ("We want")

Define:

***new*1**(X',N1',N2') :- count(X',L',N1'), iSort(L',S'), count(X',S',N2').

"We want"

MATCH

MISMATCH

Unfold; Match:

...

***new*1**(X,N1,N2) :- N1=N1'+1, count(X',L',N1'), iSort(L',S'), ins(X',S',S), count(X',S,N2).

"We have"

Insertion Sort: Difference Predicate

Introduce DIFFERENCE Predicate:

Difference between clause to be folded ("We have") and the definition ("We want")

Define:

***new*1**(X',N1',N2') :- count(X',L',N1'), iSort(L',S'), $\boxed{\text{~~count(X',S',N2')~~}}$.

"We want"

MATCH

MISMATCH

Unfold; Match:

...

***new*1**(X,N1,N2) :- N1=N1'+1, count(X',L',N1'), iSort(L',S'), $\boxed{\text{~~ins(X',S',S), count(X',S,N2)~~}}$.

"We have"

Define:

***diff*1**(X',N2',N2) :- ins(X',S',S), count(X',S,N2), count(X',S',N2').

Integer arguments

Insertion Sort: Replace

Replace: ins(X’,S’,S), count(X’,S,N2) by count(X’,S’,N2’), ***diff*1**(X’,N2’,N2)

“We have” “We want” “Difference”

Define:

***new*1**(X’,N1’,N2’) :- count(X’,L’,N1’), iSort(L’,S’), ~~count(X’,S’,N2’)~~.

“We want”

MATCH MISMATCH

Unfold; Match:

...

***new*1**(X,N1,N2) :- N1=N1’+1, count(X’,L’,N1’), iSort(L’,S’), ~~ins(X’,S’,S), count(X’,S,N2)~~.

“We have”

Insertion Sort: Replace

Replace: ins(X’,**S’**,**S**), count(X’,**S**,N2) by count(X’,**S’**,N2’), ***diff*1**(X’,N2’,N2)

“We have” “We want” “Difference”

Define:

***new*1**(X’,N1’,N2’) :- count(X’,**L’**,N1’), iSort(**L’**,**S’**), ~~count(X’,**S’**,N2’)~~.

“We want”

MATCH MISMATCH MATCH

Unfold; Match:

...

***new*1**(X,N1,N2) :- N1=N1’+1, count(X’,**L’**,N1’), iSort(**L’**,**S’**), ~~ins(X’,**S’**,**S**), count(X’,**S**,N2)~~.

“We have”

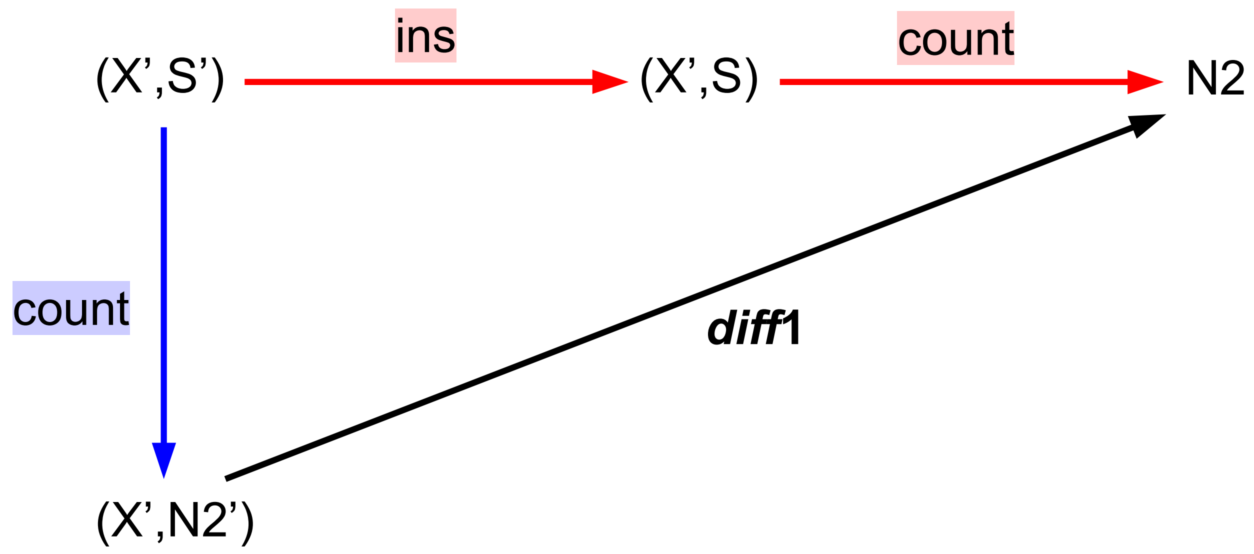
Replace:

***new*1**(X,N1,N2) :- N1=N1’+1, count(X’,**L’**,N1’), iSort(**L’**,**S’**), count(X’,**S’**,N2’), ***diff*1**(X’,N2’,N2).

Correctness of Replacement

Replace: ins(X’,S’,S), count(X’,S,N2) by count(X’,S’,N2’), ***diff*1**(X’,N2’,N2)

“We have” “We want” “Difference”



Suppose Cls U {C} → Cls U {D} by replacement.

If ***diff*1** is a function then Cls U {C} is SAT IFF Cls U {D} is SAT.

Otherwise, Cls U {C} is SAT IF Cls U {D} is SAT.

Insertion Sort: Fold

Fold: By using the definition of ***new*1**

Define:

***new*1**(X',N1',N2') :- count(X',**L'**,N1'), iSort(**L'**,**S'**), count(X',**S'**,N2').

Variant conjunctions

Unfold; Match; Replace:

...

***new*1**(X,N1,N2) :- N1=N1'+1, ~~count(X',**L'**,N1'), iSort(**L'**,**S'**), count(X',**S'**,N2')~~, ***diff*1**(X',N2',N2).

Insertion Sort: Fold

Fold: By using the definition of ***new*1**

Define:

***new*1**(X',N1',N2') :- $\boxed{\text{count(X',L',N1'), iSort(L',S'), count(X',S',N2')}}$.

Variant conjunctions

Unfold; Match; Replace:

...

***new*1**(X,N1,N2) :- N1=N1'+1, ~~count(X',L',N1'), iSort(L',S'), count(X',S',N2')~~,
***diff*1**(X',N2',N2).

Fold:

***new*1**(X,N1,N2) :- N1=N1'+1, ***new*1**(X',N1',N2'), ***diff*1**(X',N2',N2). % No lists

Insertion Sort: Final set of clauses without lists

- ```
  false :- N1=\=N2, new1(X,N1,N2).
  new1(X,0,0).
  new1(X,N1,N2) :- N1=N1'+1, new1(X',N1',N2'), diff1(X',N2',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).
  ```

  % No lists

- **diff2** is a difference predicate: diff2(X,Y,N2',N2) :- X=\=Y, ins(X,S1,S), count(Y,S,N2), count(Y,S1,N2').
- Clauses for diff1 and diff2 derived by the Elimination Algorithm, which also introduces new2, new3, new4, new5.
  ```

Insertion Sort: Computation of Model

- Eldarica proves satisfiability by computing a LIA-definable model (rewritten for legibility):

  ```
  false :- N1=\=N2, N1=N2, N2>=0.

  new1(A,B,C) :- B=C,  B>=0.
  new2(A,B) :- B = 0.

  diff1(A,B,C) :- C=B+1, B>=0.          % diff1 is a function
  diff2(A,B,C,D) :- D=C, C>=0.          % diff2 is a function

  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.
  ```

- diff1, diff2 are functions: thus, the initial and transformed clauses are equisatisfiable.

Difference Predicates and Lemma Discovery

- Eldarica model of difference predicates:

  diff1(A,B,C) :- C=B+1, B>=0.

  diff2(A,B,C,D) :- D=C, C>=0.

- Difference predicates correspond to lemmata in a proof by structural induction:

  diff1(X’,N2’,N2) :- ins(X’,S’,S), count(X’,S,N2), count(X’,S’,N2’).

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

  can be rewritten as:

  ∀((count X’ (ins X’ S’) = N2) ∧ (count X’ S’ = N2’) → (N2=N2’+1 ∧ N2’>=0))

  ∀(X=\=Y ∧ (count Y (ins X S1) = N2) ∧ (count Y S1 = N2’) → (N2=N2’ ∧ N2’>=0))

Insertion Sort: Orderedness

% Predicates of types **int**, **bool**, **list(int)**

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

ff :- 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).
```

Insertion Sort: Transformed CHCs

% Predicates of types **int**, **bool** only
% NO **list(int)**

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

ff :- 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,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).
```

Insertion Sort: Model Computed by Eldarica

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

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

- Definition of the difference predicate:

  diff(X,B,B1) :- ins(X,S1,S), ordered(S,B), ordered(S1,B1).

- Using the model, this clause corresponds to the lemma:

  $\forall((\text{ordered } S1 = B1) \land (\text{ordered } (\text{ins } X\ S1) = B) \rightarrow (B1 \rightarrow B))$.

More Sorting Programs and Properties

- Transformations done using the MAP interactive system
- Satisfiability proof an model computation done by Eldarica

	Permutation (Count)	Ordered	Length	Sum
InsertionSort	✓	✓	✓	✓
SelectionSort	✓	✓	✓	
QuickSort	✓	✗		✓
MergeSort				✓

✓ Transformation and Model Computation succeeded
✗ We gave up the transformation
Blank: We did not try

Conclusions

- CHC transformations aid verification of programs that compute on recursive datatypes;
- In the sorting examples, the Elimination Algorithm + Difference Predicate Intro transforms non-solvable (by CHC solvers) clauses into equisatisfiable solvable clauses;
- **CHC solving < (Transformation; CHC solving) ~ (Induction + CHC solving)**
- Advantage of the transformation-based approach: Separation of inductive reasoning (by transformation) from CHC solving;
- Ongoing work:
  - Mechanization (some work done);
  - Benchmarking: compare with Inductive Theorem Provers (e.g., ACL2, Clam, Leon, Isabelle).