Software Analysis and Verification 2009

Project Report

Decision Procedures for Satisfiability in the Equality Theory of Lists

Bogdan Stroe and Jonas Lindmark

under the supervsion of Prof. Viktor Kuncak

Abstract

The theory of lists plays an important role in the context of software analysis and verification. In this thechnical report we investigate two different models for this theory. The first is the recursive data type paradigm which enables us to develop a decision procedure for statisfiablity in this theory based on reducing the problem to a normal form. The second approach is to represent the problem by adapting a model used for deciding satisfiabilty of the theory of uninterpreted function symbols. We investigate each of the algorithms and show that the problem can be solved in polynomial time. Finally, we propose several extensions that would facilitate the actual application in software verification.

Contents

1. Introduction
2. Recursive Data Types
3. Problem Statement
4. Decision Procedure 1: List Normal Form Reduction
5. Decision Procedure 2: Congruence Closure on Graph Relation
6. Correctness and Completeness
7. Complexity Analysis
8. Comparison and Conclusion
9. Future Work
    a. SMT solvers
    b. Grammar Extension

Introduction

Reasoning about recursive data types such as lists and stacks is something very useful in programming today. The aim of this technical report is to provide a solution to deciding satisfiability in the equality theory of lists. A typical use case scenario would be to check that, depending on possible list variables assignments, some branch in a program could be reached. In this paper we present two procedures for determining satisfiability of conjunctions of formulas in the limited theory of lists, which can be extended to allow for more general recursive data types. We analyze and compare these procedures in relation to each other.

Recursive Data Types

The definition of a recursive data type is simply a type that contains elements of its own type. These kinds of types are often used in programming today. Perhaps the most well-known example would be the List structure where a List is defined as a concatenation of smaller Lists. The procedures presented in this paper handle a specific set of RDT's, namely when we have only one constructor. This reduces the complexity of the problem by a great degree, we discuss the implications of adding more constructors later in this paper.

Generally a recursive data type is defined by a set of constructors which generate terms of the type, a set of selectors to access the parameters of the constructed terms, a set of testers for each constructor to check whether a term was created with a specific constructor.

Throughout the paper we will use a simple List type for illustrations. This List has only one constructor, Cons. This constructor takes one element and a list and results in a new list with the element prepended to the input list. In addition we have two selectors to operate on the List, Car to access the head of a List, Cdr to access the tail of a List. In our List type the elements of the List can only be singleton variables. The grammar of the List type is:

List:=Cons(Car:Var, Cdr: List)| Var

The list [1,2,3] would be represented as Cons(1,Cons(2,Cons(3,Emptylist))). For complexity reasons the Emptylist constructor will be omitted. Instead we will abstract the end of list with a variable. It is useful to visualize this list as a tree with Cons nodes with car and cdr as children.

Problem Statement

Due to the structure of lists the problem could be represented as a special case of the SAT problem which, we will show, could be solved in polynomial time. The problem would be modeled by a set of constraints on the list of variables. The operators allowed are the ones described in the previous section. We would call one of these constraints as a *list formula*. We present the grammar of the formula:

Formula**:=** Term = Term | Term ≠ Term

Term**:=** Car(Term)|Cdr(Term) | Cons(Term,Term) | Var

We consider one of these formulas to be true if there exists and assignment for all variables present in the formula under which the formula is satisfied.

Consequently, finding a solution for the set of formulas $\{f_i | i = 1, \dots n\}$ defined by the afore-mentioned grammar would imply deciding whether there exists an interpretation of all the variables in our set of formulas under which the conjunction of all formulas evaluates to true.

$$\bigwedge_{i=1}^{n} f_i = true$$

Since each of the variables could represent any list in our theory, trying different assignments is not a feasible solution. We will attempt to solve this by modeling the formulas in a suitable way and then try to build a decision procedure on that model.

Decision Procedure 1: List Normal Form Reduction

Our procedure builds on the fact that determining satisfiability is easy when the set of formulas only contain formulas of the type $Var \neq Var$. When this is the case, we can simply go through every dis-equality and check for contradictions. We call this the normal form and present a set of rules with the purpose of reducing any set of formulas to this form. We assume that the input set of formulas is well typed.

1. Remove selectors
2. Reduce $Cons = Cons$ and $Cons = Var$
3. Remove $Var = Var$
4. Remove $Cons \neq Cons$
5. Check for contradictions

More specifically each step can be explained like this:

1. Firstly we remove the selectors from the formula. This can be easily done by introducing for each pair $Car(w)$ or $Cdr(w)$ a formula $w = Cons(wl, wr)$ if one does not exist and then evaluate the operator. In this case $Car(w)$ would be substituted by wl and $Cdr(w)$ by wr.
2. Now we want to remove $\mathrm{Cons}(a, b) = Cons(c, d)$ and $Cons(a, b) = Var$. To do this we apply two rules until they cannot be applied any more. The first rule:
   For every $Cons = Var$ or $Var = Cons$ we substitute the occurrences of that variable in the set with the cons. The second rule:
   For every $Cons(a, b) = Cons(c, d)$ we break the formula into two new formulas $\{a = c, b = d\}$ and remove the original one.

   This will reduce the original set into an equivalent set that only contains formulas of the types $Var = Var, Var \neq Var, Cons \neq Cons$.
3. Now we want to remove the $Var = Var$ formulas. This is very straight forward. If the right hand side equals the left hand side we do nothing. Otherwise we go through the set and replace all occurrences of the left hand side with the right hand side and then remove the original formula.
4. At this moment no more equalities will be generated or exist in the set. So when we encounter a $Cons(a, b) \neq Cons(c, d)$ formula we just need to check that $a \neq c \vee b \neq d$ holds by examining the names of the variables. If it does not hold we can end the algorithm here and return un-satisfiable. If it holds we simply remove the formula from the set and continue.
5. Now we have a formula in normal form and we can iterate over the inequalities and try to find contradictions. If no contradiction is found the set is satisfiable. If we find any contradiction the set is un-satisfiable.

We provide an example to depict the way the procedure works.
Consider the set:

$$\{Cons(x, y) = z, Car(w) = x, Cdr(w) = y, z \neq w\}$$

To begin with, we want to remove the selectors so we introduce $w = Cons(wl, wr)$ and evaluate the selectors which gives us

$\{Cons(x, y) = z, wl = x, wr = y, z \neq w, w = Cons(wl, wr)\}$.

Now we reduce equalities between Cons and Var and between Cons and Cons by applying rule 2. Which gives us:

$\{wl = x, wr = y, Cons(x, y) \neq Cons(wl, wr)\}$.

We apply rule 3 and get rid of the two Var-Var equalities. The set becomes just

$\{Cons(wl, wr) \neq Cons(wl, wr)\}$

Since the first arguments of the two Cons are the same(the head elements), we apply rule 4 and reach the normal form $\{ wr \neq wr\}$. This obviously leads to a contradiction in the last step and, therefore, the set of formulas is un-satisfiable.

Decision Procedure 2: Congruence Closure on Graph Relation

Open and Nelson have presented a different approach to solving this problem. More precisely, they showed how it can be reduced to the "congruence closure" problem of a relation in a directed graph.

Let G = (V, E) be a directed graph with labeled vertices, possibly with multiple edges. For a vertex v, let λ(v) denote its label and δ(v) its outdegree, that is, the number of edges leaving v. The edges leaving a vertex are ordered. For 1≤t≤δ(v), let *v[i]* denote the i*th successor* of v, that is, the vertex to which the ith edge of v points. A vertex u is *a predecessor* of v if v = *u[i]* for some i. Since multiple edges are allowed, possibly *v[t] = v[j]* for i ≠ j.

Let n be the number of vertices of G and m the number of edges of G. We assume there are no isolated vertices and therefore that n = *O(m).*

Let R be a relation on V. Two vertices u and v are *congruent under R* if λ(u) = λ(v), δ (u)= δ (v), and, for all i such that 1≤1 ≤δ (u), (u[t], *v[i]) E R. R* is *closed under congruence* if, for all vertices u and v such that u and v are congruent under R, (u, v) E R. There is a unique minimal extension R' of R such that *R'* is an equivalence relation and R' is closed under congruence; R' is the *congruence closure* of R.

After this construction we define the following algorithm:
Given a conjunction of formulas:

$$v_1 = w_1 \Lambda \dots \Lambda\, v_r = w_r \Lambda$$
$$x_1 = y_1 \Lambda \dots \Lambda\, x_s = y_s$$

Which could contain constructors and selectors Cons, Car and Cdr but also un-interpreted function symbols.

1. We build a graph G which corresponds to the set of all terms appearing in the conjunction For each term t appearing in the conjunction, let τ(t) be the vertex m G representing t For 1≤t≤r, call MERGE(τ(v_t),τ(w_t)).
2. For each node u in G labeled CONS, add vertices v, labeled CAR, and w, labeled CDR, both with outdegree i, such that v[l] = w[l] =u. Call MERGE(v,u[1]) and MERGE(w,u[2]) (That is, given a term CONS(x, y), add verttces representing CAR(CONS(x, y)) and CDR(CONS(x, y)) and merge them with the vertices representing x and y).
3. For each vertex u in G labeled Car(or Cdr), add vertex v, labeled Cons such that v[1] =u(v[2]=u). Call MERGE(v,u[1]) (or MERGE(w,u[2])).
4. For i from 1 to s, τ(x_i) is equivalent to τ(x_j), return UNSATISFIABLE. Otherwise return satisifiable.

We present the pseudo code for the Merge method (which does performs the main tasks of congruence closure).

MERGE(u, v)

1. If FIND(u) = FIND(v), then return
2. Let P be the set of all predecessors of vertices equivalent to u and P_o the set of all predecessors of vertices equivalent to v.
3. Call UNION(u, v)
4. For each pair (x, y) such that x Є P, and y Є P_o, If FIND(x) ≠ FIND(y) but CONGRUENT(x, y) =TRUE, then MERGE(x, *y).*

CONGRUENT(u, v)

1. If δ (u) ≠ δ (v), then return FALSE
2. For 1 ≤ t ≤ δ(u), if FIND(u[t]) ≠ FIND(v[t]), then return FALSE
3. Return TRUE

UNION(u,v)

1. Unify the classes of vertices u and v

In the follwing we will present the correspondence of the above algorithm to our list problem .

First, we construct the grah G. Each term in our set of formulas represents a node in G. A term *a* is a predecessor of another term *b* if *a* is a function of *b.* We must however not forget that we are not just dealing with uninterpreted functions but with list constructors and selctors. Therefore we must introduce additional nodes. For every *Cons(a,b)* term we shoud introduce *Car(Cons(a,b))* which has the same equivalence clas as *a* and *Cdr(Cons(a,b))* which has the same equivalence class of *b*. Also, for each *Car(a)* and *Cdr(a)* terms we introduce a new term *Cons(Car(a),Cdr(a))* which has the same equivalence class as a. These new terms/nodes are just to model the idea that the concation of the head of a list and tail of the list is the list itself.

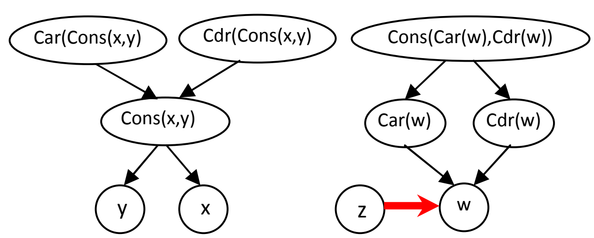
The Relation R is determined by the equalities in the given set of formulas. For each formula *a=b* we have that $(a, b)\epsilon R$. The next step is to call the MERGE procedure for each pair *(a,b)* .

The final step of the decision procedure is to iterate throught each dis-equality in the set of formulas and check if the two terms have the same equivalence class. In case they do, then we can safely say that the set of formulas is unsatisfiable. Otherwise, we can conclude that the problem is satisfiable.

For example let F be the set of formulas.

$F = \{ Cons(x, y) = z, Car(w) = x, Cdr(w) = y, z \neq w\}$

We construct the graph G according to the algorithm presented earlier:



Due to the realation inferred by F we have that term x is in the same equivalence class as car(w) and term y is in the same equivalence class as cdr(w). By congruence closure we can infer that also cons(x,y) is in the same equivalence class as cons(car(w),cdr(w)). However the relation implied by F claims that z is in the same relation as cons(x,y) and we know that cons(car(w),cdr(w)) is actually w. This would imply by transitivity that w is the same equivalence class as z. This yields that F is unsatisifiable since it contains the dis-equality between z and w.

Correctness and Completeness

We will follow a simple approach in showing that the fist algorithm is correct. We will show that each step we perform on our set of formulas preserves satisfiabilty. Also we will show that if we start with an unsatisfiable formula it will remain unsatisfiable after we apply any of the steps.

1. Removing selectors
2. Replacing cons variables
3. Splitting cons=cons formulas
4. Substituting variables
5. Removing cons ≠ cons formulas

We will also show that we will always reach a normal form for our set of formulas. All of these steps are made under the assumption that we have typed list variables.

1. Suppose there exists an assignment for which the set of formulas F is satisfiable. We can construct an assignment for the F-noSelectors set of formulas that result after step 1. We simply extend each of the initial assignment such for the unassigned variables in the following way. Every variable vl is assigned to the head of list v and to every variable vr is assigned the tail of the list v. This is obviously satisfiable since we represent the same structure as the initial set of formulas. Conversely, if the initial set is unsatisfiable. Then also the new set of formulas is unsatisfiable. Otherwise, suppose F-noSelectors has a variable assignment under which it is true. In this case we could construct an satisfying assignment also for the initial set of formulas by just removing the variable introduced. For each new formula v=cons(vl,vr) we replace all instances of vl and vr by car(v) and cdr(v) everywhere in our set of formulas and remove both vl and vr from our assignment. We would obtain the original set F and a satisfying assignemnet for it, thus, we have reached the contradiction.
2. We interpret this step as a substitution of a variable in all our formulas. Therefore, in the initial intial conjuction of formulas, the existential quantifier over that variable turns itself in a universal quantifier. It is trivial to show that: $\exists v \exists u_1 \ldots \exists u_n . v = cons(a, b) \Lambda f(v, u_1 \ldots u_n)$

is equivalent to

$\forall v \exists u_1 \dots \exists u_n . \left[{}^{cons(a,b)}/_{v}\right] f(v, u_1 \dots u_n)$

Therefore, if the first formula is satisfiable also the second one is. The same holds for the unstatisfiable case.

3. Again replacing a cons=cons formulas will result in an equivalent sets of formulas. This results from the fact that $cons(a,b) = cons(c,d)$ is equivalent to $a = b \wedge c = d$.
4. The correctness proof for this step is identical to step 2 .
5. We have two situations here. In the first case, we are trying to remove a formula $cons(a,t1) \neq cons(a,t2)$. This formula is equivalent to $t1 \neq t2$. In the second case we have $cons(a,t1) \neq cons(c,t2)$. At this point we can completely remove the formula. This is becasuse since the heads of the lists are different and the dis-equality can always be satisfied. It is imporntant note that, due to step 4, at this point if we would have had information that a=c there would be just one variable for this element and we would be in the first case. Another important aspect is that we cannot have a cons term as the first parameter of another cons term but only a variable term because the head of a list cannot be a list but only an element. This is a direct implication to that fact that we are dealing with typed lists.

To prove completeness we must show that this algorithm ensures that the normal form is always reached. As pointed out previously, the normal form we reach is a set of dis-equlities between variables. The first rule makes sure that we have no selectors in the terms of our set of formulas. The next two steps are run over the obtained set exhaustively untill neither is applicable anymore. By defintion of these rules, this means that after this part of the algorithm we will not have any other formulas except for the ones of type $cons(a,b) \neq cons(c,d), a = b \; or \; a \neq b$. Also this step terminates too because with each cons-variable repalcement we remove completely one variable from our set of formulas. Since we have finite number of variables and finite number of formulas this shows completeness of these 2 steps.

Step 4 is equivalent to building equivalence classes for each varible based on the equality formulas of type $a = b$. Again, this will always terminate because at each step we remove at least one variable from our formulas.

The final step reduces the set of formulas to the normal form. It is important to notice that by replacing formula $cons(a,b) \neq cons(a,b)$ we will never introduce equalities but only dis-equalities. This ensures that the previous steps do not need to be applied again after this step. At this point we will have only dis-equalities between equivalence clases. We will decide that the formula is unsatisfialbe only if we encounter a formula of type $a \neq a$. If we don't find such a formula we canc safely say that the inital set of formulas is satisfiable due to the correctness of the algorithm. Since this step will also alway terminate we can conclude that the algorithm is complete.

The correctness and compalteness of the second algorithm shown quite easily by costructing of satisfying interpretations just as we have done for the first algorithm and is presented in detail in [2].

Complexity

The complexity of the second algorithm is shown to be $O(m^2)$ where m is the number of edges in the resulting Graph. The proof can be found in [2].

We shall however analyze the complexity of the first procedure.

The algorithm is divided into four steps:

1. Selector removal
2. Reducing $Cons = Cons$ and $Cons = Var$
3. Removing $Var = Var$
4. Removing $Cons \neq Cons$
5. Checking for contradictions

Selector removal works by iterating over the set and for each selector found we introduce a Cons construction for that variable and attempt to evaluate all selectors on that variable in the set. This gives us complexity of O(|Terms|x|Terms|).

The second step we can visualize as a loop:

```
while(changeMade){
  removeConsCons();
  removeConsVar();
}
```

The removeConsCons will search through the set for a Cons = Cons formula and then reduce it. The reduce step is constant in complexity so the total complexity of removeConsCons is O(|Terms|). The removeConsVar operation will each time it is applied reduce the amount of variables by one. Hence it can at most be applied as many times as the amount of variables we have in the set. Applying the rule once means that we have to go through every term in the set to search for possible variables to substitute. This gives us the complexity of O(|Vars|x|Terms|).

The $Var = Var$ replacing procedure is quadratic in terms of the amount of terms since we go through each formula and for every $Var = Var$ we have to search through the set again for the substitution. O(|Terms|x|Terms|).

Removing $Cons \neq Cons$ works by iterating over the set to find the $Cons \neq Cons$ formulas and for each of them search for the head equalities and if it exists also search for tail equalities. This gives us complexity of O(|Terms|x|Terms|).

The final contradiction step works by for each $Var \neq Var$ formula check whether the right hand side is equal to the left side. This is linear in terms of the amount of terms in the set, O(|Terms|).

So in total we have $O(|Terms|^2 + |Terms| + |Terms| * |Vars|)$ which is the same as $O(|Terms|^2 + |Terms| * |Vars|)$.

NP completeness

Earlier in the paper we have stated that including the empty list in the List type makes the satisfiability problem significantly harder. The problem actually becomes NP complete. There is a quite simple reduction to 3CNF-SAT. To begin with, if we add the Nil constructor to the List we will have to define the behavior of the selectors on that List. The most reasonable behavior is that evaluating Car(Nil) and Cdr(Nil) will result in Nil. Also we add that $Cons(Nil, Nil) \neq Nil$, although it does not make sense to write $Cons(Nil, Nil)$ since the first argument must be an element but since we defined $Car(Nil) = Nil$ it does work.

Given a 3CNF-SAT problem consisting of $p_1 \dots p_n$ truth variables and a conjunction F of 3-element clauses containing $p_1 \dots p_n$ we construct the conjunction G of variables $x_1, y_1, \dots, x_n, y_n$ and we add the initial construction of G:

$\{Car(x_i) = Car(y_i), Cdr(x_i) = Cdr(y_i),$
$x_i \neq y_i\}$ where $i = 1 \dots n$

This would mean that x_i and y_i cannot both be Nil. Since the heads and the tails of them would be equal, one will have to be Nil and one would be $Cons(Nil, Nil)$. We define that for p_i to be true then x_i must be Nil and conversely p_i is false then $y_i = Nil$.

Now it is easy to model the 3-CNF problem into a List satisfiability problem. For example, say that we want to model the clause $p_1 \vee {\sim}p_2 \vee p_3$ which is equivalent to $x_1 = Nil \vee y_2 = Nil \vee x_3 = Nil$. Rewriting this into a conjunction we get ${\sim}(y_1 = Nil \wedge x_2 = Nil \wedge y_3 = Nil)$ This would give us the List formula $Cons(y_1, Cons(x_2, y_3)) \neq Cons(Nil, Cons(Nil, Nil))$.

This shows that the problem is NP hard, it is easy to see that given a solution we can verify that it is correct in polynomial time which tells us the problem is also NP complete.

Conclusion

We have presented the technical aspects of two approaches that could solve the decision problem on satisfiability of quantifier free equality theory of lists. These two algorithms differ in the approach of handling this problem but are able to generate a solution with similar complexity. Unlike the general SAT problem, the solution could be found in polynomial time.

We can distinguish a specific particularity that differentiates the approach of the two algorithms. They key step in the normal form reduction algorithm is to infer relationships between the children of two terms based on the relationship of the parent terms. In the second algorithm we do the opposite. We merge equivalence classes of parents based on the equivalence classes of the children. However, the operation count is similar in both cases because for the second algorithm we introduce as many nodes in our graph as new variables introduced in the set of formulas by the "remove selectors" of the first algorithm. Intuitively, one can notice that the first algorithm does some extra computation steps in case the formulas have many selectors. This aspect introduces a computation overhead also for the subsequent steps of the first algorithm. In the graph representation each selector is linked to its argument by just one edge which makes the congruence closure step run faster. We can conclude that each procedure would run faster than the other on some specific type of input formulas.

Nonetheless it is difficult to provide an accurate efficiency comparison because it is difficult to come up with an exhaustive set of input examples. A future project could consist of determining the properties of a given set of formulas which could recommend the most efficient of the two decision procedures for this particular case.

Future Work

SMT Solvers

Satisfiability Modulo Theories is the problem of determining satisfiability of a conjunction of formulas with respect to combinations of background theories. This extension to the problem we have discussed is very useful in reasoning about programs since in most practical applications more than one theory is used. There exists a couple of different algorithms for solving SMT problems. For example the DPLL(T) architechture.

DPLL(T)

DPLL stands for Davis, Putnman, Logeman, and Loveland which are the creators of the original DPLL algorithm. They proposed a backtracking algorithm for solving the CNF-SAT problem, this algorithm is widely used because of its efficiency. The (T) stands for "modulo Theory" which means that the DPLL(T) determines satisfiability of CNF formulas in the theory T, in the case of EUF the theory T would simply be the theory of equality. To produce a DPLL(T) system one has to instantiate a general DPLL(X) with a $Solver_T$ for the theory T. This $Solver_T$ must be able to handle conjunctions of formulas in the theory T and acts as an interface between the general DPLL(X) and the theory T.

Combining Theories

Combining decision procedures for different theories is very useful. Nelson and Oppen designed an algorithm called the *Nelson-Oppen combination method[4]* which exists in two variations, deterministic and non-deterministic. This method takes two or more theories and decision procedures for the quantifier free fragments of these theories and produces a system that solves the quantifier free version of the union of these theories.

Extending the grammar

Another possible extension would be to have more selectors and constructors in our list theory. The first algorithm would be more difficult to adapt since it would require modeling also disjunctions of formulas. The real problem stems from the fact that, when we try to simplify the formulas, we don't know which constructor was used to create that list variable. We would have to introduce some new rules for this and assign to each list variable a set of possible constructors that could have been used to generate it[1]. As we process the list of formulas we should narrow down this set for each of the variables. In some cases we will have to test disjunctions of sets of possible constructors. Of course, this would introduce an additional overhead since we would have to explore different assignments at the same time. It would be necessary to have a strategy that splits the set possible assignments so that it can be explored in an efficient way.

The second algorithm would run into the same problem. However since the general algorithm uses uninterpreted function symbols it would be easier to model. The new constructors would just be new functions in our theory. However, we would have to model in terms of edges in our graph the semantics of these new constructors and selectors with respect to lists. In terms of implementation this approach seems less error prone due to the fact that we do not have to modify significantly the original algorithm.

References

[1]An Abstract Decision Procedure for Satisfiability in the Theory of Recursive Data Types
Clark Barretta, Igor Shikaniana, Cesare Tinellib

[2]Fast Decision Procedures Based on Congruence Closure
Greg Nelson, Derek C. Oppen

[3]On the Theory of Structural Subtyping
Viktor Kuncak, Martin Rinard

[4]The Calculus of Computation- Decision Procedures with Application to Verification
Aaron R. Bradley, Zohar Manna