An Eager SMT Solver for Algebraic Data Type Queries

Amar Shah
University of California, Berkeley

1 Introduction and Motivation

Algebraic Data Types (ADTs) are a programming construct classically found in functional programming languages but are increasingly found in all kinds of modern languages. ADTs are a convenient generalization of structures like enumerated types, lists, and binary trees.

A natural problem is the satisfiability of formulas over the theory **ADT**. This has applications in modelling languages [Milner 1978], proof assistants [Gonthier 2005] and program verification [Bjørner et al. 2013]. While the need to reason about ADTs have grown, the techniques to do so have not.

Satisfiability Modulo Theory (SMT) extends the Boolean Satisfiability (SAT) problem to include additional theories, in this case **ADT**. Most SMT solvers for **ADT** apply the same *lazy* approach that use a theory solver [Oppen 1980] in a loop with a SAT solver.

We propose a fundamentally different approach: an *eager* solver for **ADT** satifiability modulo theory (SMT) queries via a quantifier free reduction to Equality and Uninterpreted Functions (**EUF**) SMT queries. This work presents:

1. A reduction from **ADT** to **EUF**
2. An SMT solver Algaroba that implements this reduction in order to solve Quantifier-Free ADT queries
3. A description of non-trivial optimizations in Algaroba
4. An evaluation of Algaroba in comparison to state-of-the-art SMT solvers

2 Background

A theory is a set of sentences in a formal mathematical language. An SMT solver determines if sentences are satisfiable with a background theory. For example **EUF** is a theory with only symbolic constants, applications of functions, and basic logical connectives (like $\wedge, \vee, \neg$). **ADT** is our theory of Algebraic Datatypes. See Barrett et al. [2017] for the full formal treatment of these theories.

Our solver takes a quantifier free formula ψ in **ADT** , reducing to quantifier free in **EUF** and then applying an SMT solver to get a **sat** or **unsat** result:

> **Definition 2.1.** *A theory T **reduces** to a theory R if there is a computable m s.t. $T \models \psi \leftrightarrow R \models m(\psi)$*

We will build our theory of **ADT** with functions called constructors, selectors, and testers. Here is an example of how we would define the list ADT:

```
(declare-datatype List ((Nil) (Cons (head Int) (tail List)) ))
```

The definition uses two constructors: `Nil` and `Cons` which are the two possible ways to build a `List`. `Nil` takes no inputs and outputs a `List`. `Cons` is a function that takes an `Int` and a `List` and outputs a `List`. Each corresponding constructor has a set of selectors. `Nil` has no selectors, but `Cons` has selectors given by `Head` and `Tail`. These can be thought of ways to de-construct a list, i.e. get back to the terms that we used to build a `List`. The definition implicitly defines two testers: `is_Nil` and `is_Cons`. These functions are from `Lists` to `True` or `False` and essentially tell you how a given list was constructed.

> **Definition 2.2** (Algebraic Data Type)**.** *An instance of an ADT $\mathcal{A}$ is a tuple consisting of:*
>
> - *A set $\mathcal{A}^S \subseteq \mathcal{S}$ of sort symbols containing **Bool***
> - *A distinguished finite set of constructors $\mathcal{A}^C \subset \mathcal{F}$, where each constructor has a sort σ and arity l for a constructor $f : \sigma_1 \times ... \times \sigma_l \rightarrow \sigma$*
> - *A distinguished finite set of selectors $\mathcal{A}^S \subset \mathcal{F}$, such that there are l distinct selectors $f^1, ..., f^l$ for each constructor $f \in \mathcal{A}^C$ with arity l.*
> - *A distinguished finite set of testers $\mathcal{A}^T \subset \mathcal{F}$ and a bijection $p : \mathcal{A}^C \rightarrow \mathcal{A}^T$ which sends $f \mapsto is_f$*
>
> *Additionally, we want the requirement that restricting our ADT $\mathcal{A}$ to just the base terms and constructors is well-sorted, i.e. there are no circular dependencies in how we define each term.*

3 Approach

The idea behind our reduction will be to encode the axioms of **ADT** in the language of **EUF**. We cannot do this directly, since these axioms have universal quantifiers. Solving theories with universal quantifiers is expensive and is not supported by many SMT solvers. Instead ,we will only solve **ADT** queries on quantifier free formulas by reducing them to **EUF** quantifier free formulas. We use a technique called "blasting": we will only instantiate our axioms over terms that appear in the query.

For a formula ψ in **ADT** we will reduce this to $\psi^* \wedge \phi_1 \wedge ... \wedge \phi_m$ where $\{\phi_i\}$ are additional axioms we must satisfy and ψ^* in **EUF** is a modified version of ψ created by the rules:

$$\text{A . } f(t_1...t_l) = t \implies f(t_1, ...t_l) = t \wedge is_f(t) \wedge \bigwedge_{i=1}^{l} f^i(t) = t_i$$

$$\text{B . } f^j(t) = t_j \implies f^j(t) = t_j \wedge \bigvee_{g \in \{f_1, ... f_n\}} [\exists t_1, ..., t_l [g(t_1, ..., t_l) = t \wedge \bigwedge_{j=1}^{l} g^j(t) = t_j]]$$

C. $is_f(t) \implies \exists t_1, ..., t_l [f(t_1, ..., t_l) = t \wedge \bigwedge_{j=1}^{l} f^j(t) = t_j]$

These rules ensure that constructors, testers, and selectors all behave well with one another. To create our axioms ϕ_1, ... ϕ_m, we blast over the set T which is the set of all variables that appear in our query. For $t \in T$ we want:

1. For any tester in $\{is_{f_i}\}_{1 \le i \le |C_\sigma|}$, we add the axiom $\phi :=$ $\bigvee_{i=1}^{|C_\sigma|} [is_{f_i}(t) \wedge \bigwedge_{j=1, j \ne i}^{|C_\sigma|} \neg is_{f_j}(t)]$

This axiom ensures that each variable satisfies exactly one tester. This reduction is almost correct, except we need to ensure the "well-sortedness" property of **ADT**. In Section (4), we define the correct set T so that we are considering all possible cyclic relationships between terms.

4 Reduction

We can take an example query over lists:

```
   (and (= (tail y) x) (= (tail x) y))
```

Clearly this is unsatisfiable since no well-sorted structure could have x and y as tails of each other. This can be generalized to even more variables, take variables $x_1, ..., x_k$: `List`

```
(and (= (tail x_1) x_2) ...
    (= (tail x_{k-1}) x_k) (= (tail x_k) x_0))
```

Thus, we need an axiom to ensure this is unsatisfiable in our reduced query. Let k be the number of variables that appear in the input query. Define $T_0 = \{t : t \text{ is a term in } \psi\}$ and for $i = 0, ..., k-1$, define $T_{i+1} = \{s | t \in T_i$ and exists a selector f^j s.t. $is_f(t) \wedge f^j(t) = s\}$. Then we define $T = \bigcup_{i=0}^{k} T_i$. We call s a subterm of t of depth i if s can be obtained via a sequence of i selectors applied to t. Now we can introduce a second axiom that encodes this well-sortedness constraint into our reduction:

2. For each $t, s \in T$ where we know that s is a subterm of t, we add the axiom $s \ne t$

Theorem 4.1. *Say ψ is an **ADT**-formula that is in flat NNF form. If we define T as above, then* **ADT** $\models$ $\psi \leftrightarrow$ **EUF** $\models \psi^* \wedge \phi_1 \wedge ... \wedge \phi_m$ *where we compute ψ^* from ψ using Rules A, B, C and $\phi_1, ...\phi_m$ using Axioms 1 and 2. This is a reduction as in Definition (2.1)*

Proof Sketch. $\underline{\rightarrow}$: If the **ADT** $\models \psi$, then **EUF** $\models \psi^* \wedge \phi_1 \wedge ... \wedge \phi_m$ since we only introduce constraints with the axioms of **ADT**

$\underline{\leftarrow}$: Since **EUF** $\models \psi^*$, for every variable x in ψ, it must be that there is exactly one tester is_f such that $\psi^* \rightarrow is_f(x)$. by Axiom (1) and one constructor f such that x is in the codomain of f by Rule (C).

Then we can apply each selector $f^1, ..., f^l$ to get l total subterms. We keep applying selectors to each of these subterms until we have considered all subterms up to depth k. We may reach subterms that appear in our input query ψ. However, by Axiom 2 of our reduction, we know that in **EUF**, these subterms cannot be equal to our original term.

Note that it does not really matter what these subterms of depth more than k are, since our original query ψ cannot say anything about relations of depth more than k(since it is a flat, NNF formula). Thus, we can let deeper subterms of x be constants.

We do the same for all other variables in ψ and we have created a satisfying assignement for ψ in **ADT** □

5 Complexity

This reduction can create an exponential blowup in the size of the query. We know the depth k is at most linear in the size of the query, since it is the number of variables (and converting to NNF and flattening our query will only create a linear blowup). However, take a term x of the `tree` type

```
type tree =
  | Leaf
  | Node of {left: tree; right: tree}
```

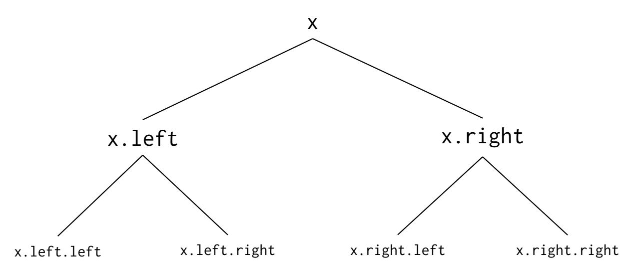


Figure 1. For a term x: `tree`, the number of selector applications up to depth k is $2^{k+1} - 2$, thus giving us an exponential blowup in the number of terms

In Fig 1 the `tree` datatype has a constructor `Node` that is constructed from two instances of `tree`.

6 Algorithm

This exponential blowup is problematic since the depth k we compute is a large overapproximation. What if we instead introduced these acyclicality axioms *iteratively*. In doing so, we hope to combine the advantages of both eager and lazy solvers.

Recall, Thm 4.1 states that $(\psi, \textbf{ADT})$ is **sat** $\leftrightarrow$ $(\psi^* \wedge \phi_1 \wedge ... \wedge \phi_m, \textbf{EUF})$ is **sat**, where we compute $\psi^* \wedge \phi_1 \wedge ... \wedge \phi_m$ from ψ using rules A, B, C and axioms 1, 2, 3.

However, what if had $\psi^* \wedge \phi_1 \wedge ... \wedge \phi_l$ for some $l < m$. For example if we did not instantiate all of the axioms from acyclicality axioms defined in axiom 2. In this case we still have:

$$(\psi^* \wedge \phi_1 \wedge ... \wedge \phi_l, \textbf{UF}) \text{ is } \textbf{unsat} \rightarrow (\psi^* \textbf{ADT}) \text{ is } \textbf{unsat}.$$

Algorithm 1 Algaroba(ψ)

$k \leftarrow$ Number of variables in ψ
$\tilde{\psi} \leftarrow$ Apply rules A, B, & C and axiom 1 to ψ
for $i = 0$ **to** $i = k$ **do**
  $\phi_1 \wedge ... \wedge \phi_n \leftarrow$ apply axiom 2 up to depth i
  match (*UF-SMT-Solver*($\psi^* \wedge \phi_1 ... \phi_n$)) **with**
    case unsat:
      return unsat
    case sat:
      $\alpha_1 \wedge ... \wedge \alpha_p \leftarrow$ Restricting the size of our model to i
      match (*UF-SMT-Solver*($\psi^* \wedge \phi_1 ... \phi_m \wedge \alpha_1 ... \alpha_p$)) **with**
        case unsat:
          return unsat
        case sat:
          continue

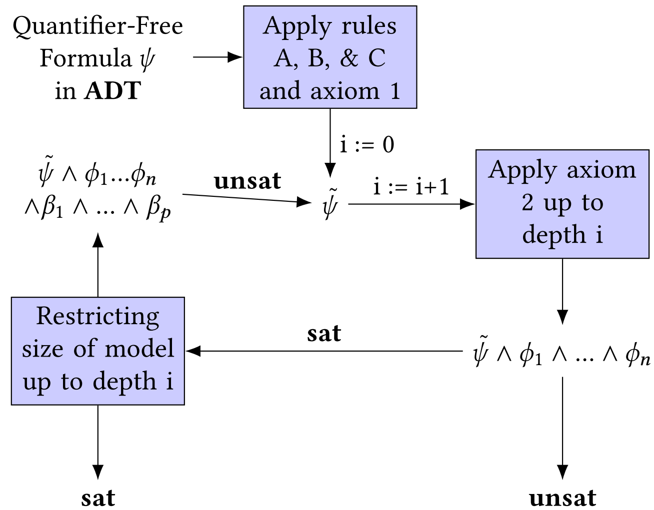


Figure 2. Pseudocode and illustration showing reduction architecture.

Thus, if we evaluate this partially reduced query $\psi^* \wedge \phi_1 \wedge ... \wedge \phi_l$ using a **EUF** solver, we can trust **unsat** results, but not the **sat** result. This provides an idea for how to build a new solver, we can slowly increase the depth, introducing acyclicality axioms, and checking for **unsat**.

In order to iteratively check for **sat** results we assume the model is a fixed size i at each step i.

We implement a prototype of our approach in approximately 2900 lines of OCaml code using the Z3 API as *UF-SMT-Solver*. We call this prototype Algaroba and show a high-level architecture diagram in Fig. 2.

7 Optimizations

7.1 Counting a number k for each individual ADT $\mathcal{A}$

In practice, we do not use the same k for all ADTs, rather for each ADT $\mathcal{A}$, we compute a specific $k_{\mathcal{A}}$. In order to compute this, we need to keep track of which ADTs reference each other, for example:

```
type nat =
  | succ {pred: nat}
  | zero
type list =
  | Null
  | Cons of {head: tree; tail: list}
  | node of {children: list}
  | leaf of {data : nat}
```

Listing 1. Type declarations for tree, list, nat

Here, since `tree` is the declaration of a tree with an arbritrary number of children we use a `list` to store its children. Thus, `list` and `tree` refer to one another. We can look at a graph of these references:

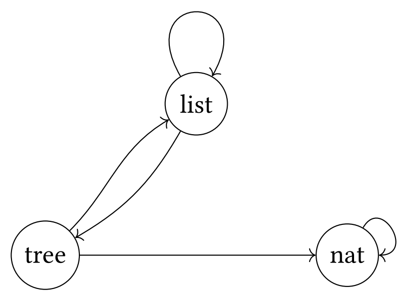
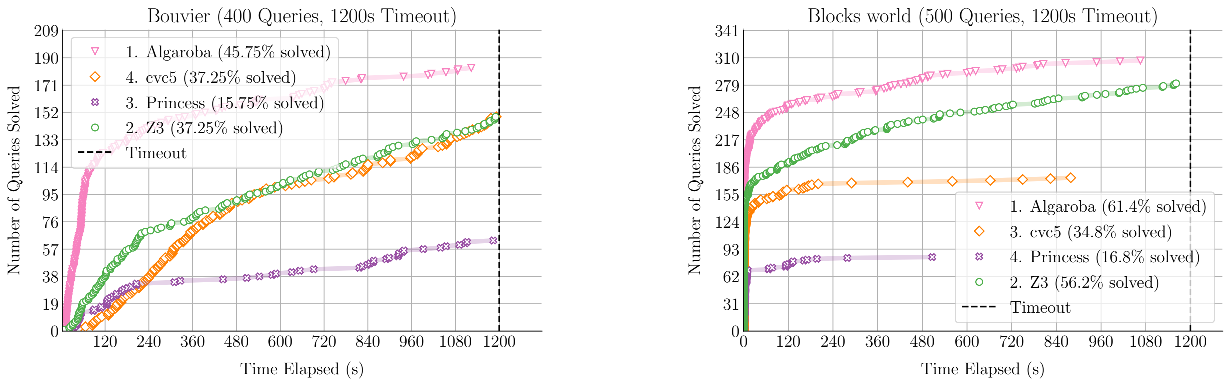


Figure 3. The types from Listing 1 where the arrow from $\texttt{type}_1$ to $\texttt{type}_2$ represent that there is a selector $s : \texttt{type}_1 \rightarrow \texttt{type}_2$

For an ADT $\mathcal{A}$, we only care about sequences of selectors $s_1, ..., s_m$ such that $s_m \circ ... \circ s_1 : \mathcal{A} \rightarrow \mathcal{A}$ since these are the sequences of selectors that will create cycles. Thus, we only care about an ADT $\mathcal{B}$ if it is in the same SCC as $\mathcal{A}$, since only then can $\mathcal{A}$ and $\mathcal{B}$ be in the same cycle.

Thus, we can store the above graph as an adjacency list, and then run a Strongly Connected Connected Component (SCC) Algorithm. Above, we have that there are two SCCs: (1) `list` and `tree`; and (2) `nat`. We use Tarjan's SCC algorithm which runs in linear time in the number of vertices and edges (for us the number of ADTs and selectors) [Tarjan 1972].

Note that if there is no path from a vertex for $\mathcal{A}$ to itself in the graph (including self loops), then we can set $k_{\mathcal{A}} = 0$ since $\mathcal{A}$ is not an inductive datatype.



(a) Bouvier benchmark set.

(b) Blocks world benchmark set.

Figure 4. Number of queries solved (y) in less than x seconds for Bouvier and blocks world benchmark sets using a 1200s timeout. Higher (more queries solved) left (in less time) points are better. The legend lists the contribution rank and percentage of queries solved for each solver. Algaroba solves the most queries and achieves the highest contribution rank for both sets.

8 Evaluation

Our solver Algaroba takes inputs in the SMT-LIB language and includes a number of simple optimizations, like hash-consing [Ershov 1958], incremental solving, and theory-specific query simplifications. All experiments are conducted on an Ubuntu workstation with nine Intel(R) Core(TM) i9-9900X CPUs running at 3.50 GHz and with 62 GB of RAM. All solvers were given a 1200 second timeout on each query to be consistent with SMT-COMP. We compare it to three state-of-the-art solvers in this space cvc5, Z3, and Princess. ion since it is the most related approach.

Our evaluation covers two existing benchmark sets from SMT-COMP, the standar competition for SMT solvers. One is originally from Bouvier [2021] and one is originally from Shah et al. [2024] (which we refer to as blocksworld).

We time the execution of Algaroba, cvc5, Princess, and Z3 on all queries in all three benchmark sets. Algaroba clearly outperforms the rest, solving 8.5 percentage points more queries than the second best on Bouvier and 5.2 percentage points more queries than the second best on blocksworld. We conclude favorably, that Algaroba's novel solving technique leads to better performance on real world queries.

9 Acknowledgments

Thank you to Federico Mora Rocha for presenting me with this problem, mentorship, and support. I would like to also acknowledge Professor Sanjit Seshia and the University of California, Berkeley's *Learn and Verify* lab for their support with this research. This is an extension work done in collaboration with Federico Mora and Sanjit Seshia [Shah 2023; Shah et al. 2024]

References

Clark Barrett, Pacal Fontaine, and Cesare Tineli. 2017. The SMT-LIB Standard Version 2.6. https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf.

Nikolaj S. Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. 2013. Higher-order Program Verification as Satisfiability Modulo Theories with Algebraic Data-types. *CoRR* abs/1306.5264 (2013). arXiv:1306.5264 http://arxiv.org/abs/1306.5264

Pierre Bouvier. 2021. The VLSAT-3 Benchmark Suite. *INRIA Technical Report 516* (December 2021).

A. P. Ershov. 1958. On Programming of Arithmetic Operations. *Commun. ACM* 1, 8 (aug 1958), 3–6. https://doi.org/10.1145/368892.368907

Georges Gonthier. 2005. A computer-checked proof of the Four Colour Theorem.

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Derek C. Oppen. 1980. Reasoning About Recursively Defined Data Structures. *J. ACM* 27, 3 (jul 1980), 403–411. https://doi.org/10.1145/322203.322204

Amar Shah. 2023. An Eager SMT Solver for Algebraic Data Type Queries. (July 2023). Programming Languages Design & Implementation Student Research Competition (PLDI SRC).

Amar Shah, Federico Mora, and Sanjit A. Seshia. 2024. An Eager Satisfiability Modulo Theories Solver for Algebraic Datatypes. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (Eds.). AAAI Press, 8099–8107. https://doi.org/10.1609/AAAI.V38I8.28649

Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. https://doi.org/10.1137/0201010 arXiv:https://doi.org/10.1137/0201010