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]. We propose an eager solver for **ADT** satifiability modulo theory (SMT) queries via a quantifier free reduction to Equality and Uninterpreted Functions (**EUF**) SMT queries. This improves on existing solvers [Hojjat and Rümmer 2017] [Kostyukov et al. 2021] for **ADT** since it can be used in tandem with any SMT solver that solves **EUF** queries and it can be easily used in a high performance computing setting [Pimpalkhare et al. 2021].

2 Background

A theory is a set of sentences in a formal mathematical language. A satifiability modulo theory (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** that is in flat, NNF form, 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. $T \models \psi \leftrightarrow R \models m(\psi)$*
*A formula ϕ is **flat** if function symbols only occur in equations of the form $x = f(x_1, ..., x_n)$ where $x, x_1, ..., x_n$ are variables.*
*A formula ϕ is in **Negation Normal Form (NNF)** if the only Boolean operators are conjunction, disjunction, and negation only applied to atomic formulas.*

Note that any query can be reduced to a flat NNF form.

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 $f : \sigma_1 \times ... \times \sigma_l \to \sigma$*
- *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 $p : \mathcal{A}^C \to \mathcal{A}^T$ 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 $\{is_{f_i}\}_{1\leq i\leq |C_\sigma|}$, we add the axiom $\phi := \bigvee_{i=1}^{|C_\sigma|} [is_{f_i}(t) \wedge \bigwedge_{j=1, j\neq i}^{|C_\sigma|} \neg is_{f_j}(t)]$

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 in fact be generalized to even more variables. Thus, we need an axiom to encode this property into our reduction.

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 $T_{i+1} = \{s | t \in T_i \text{ and exists a selector } f^j \text{ s.t. } is_f(t) \wedge f^j(t) = s\}$. Then we define $T = \bigcup_{i=0}^{k} T_i$. 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 $s \neq t$

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. $\underline{\rightarrow}$: 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).

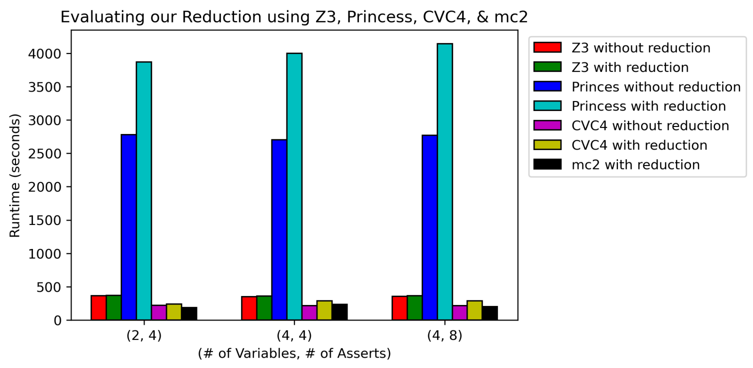
Then we can apply each selector $f^1, ..., f^l$ to get l total subterms. We keep applying selectors to each of these subterms until we have considered all subterms up to depth k. We may reach subterms that appear in our input query ψ. However, by Axiom 2 of our reduction, we know that in **EUF**, these 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 Evaluation & Future Work

We have a *Python* implementation of this reduction for flat NNF for `List` with `Int` (and `Real` for *mc2*) in less than 200 lines of code. We fuzzed random inputs and tested this reduction on common edge cases to verify correction. To test runtime we ran four popular SMT solvers *z3* [de Moura and Bjørner 2008], *Princess* [Rümmer 2008], *CVC4* [Barrett et al. 2011], and *mc2* [de Moura and Jovanović 2013] on 10,000 randomly generated queries on a M1 8-core Mac with 8 GB of RAM:



Runtime of our Reduction on *Z3*, *Princess*,a nd *CVC4*			
(Vars, Asserts)	(2, 4)	(4, 4)	(4, 8)
# SAT	2141	526	3154
# UNSAT	7859	9474	6846
Z3 Prereduction (sec)	363.67	353.01	356.36
Z3 Post Reduction (sec)	368.20	359.85	365.89
Princess Pre-Reduction (sec)	2777.6	2700.6	2771.8
Princess Post-Reduction (sec)	3689.9	4000.4	4141.4
CVC4 Pre-Reduction (sec)	219.42	218.80	215.97
CVC4 Post-Reduction (sec)	240.97	288.43	288.00
mc2 Post-Reduction (sec)	189.54	234.27	202.37

We do worse on the three solvers with built-in ADT solving using our reduction. However, the strength of our approach comes from the fact that once we do the reduction any SMT solver with support of **EUF** can solve the query. Indeed, *mc2* (the one solver without support for **ADT**) is the fastest on 2 of the 3 tests. We expect that with certain optimizations and used in a high performance computing setting, we can get an even faster solver.

6 Acknowledgments

Thank you to Federico Mora and Sanjit A. Seshia for presenting me with this problem and mentoring me.

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.

Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. https://doi.org/10.1007/978-3-642-22110-1_14

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

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

Leonardo de Moura and Dejan Jovanović. 2013. A Model-Constructing Satisfiability Calculus. In *Verification, Model Checking, and Abstract Interpretation*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–12.

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

Hossein Hojjat and Phillipi Rümmer. 2017. Deciding and Interpolating Algebraic Datatypes by Reduction. (2017). https://ieeexplore.ieee.org/document/8531279

Yuri Kostyukov, Dmitry Mordvinov, and Grigory Fedyukovich. 2021. Beyond the elementary representations of program invariants over algebraic data types. (2021). https://dl.acm.org/doi/10.1145/3453483.3454055

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

Nikhil Pimpalkhare, Federico Mora, Elizabeth Polgreen, and Sanjit A. Seshia. 2021. MedleySolver: Online SMT Algorithm Selection. In *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12831)*, Chu-Min Li and Felip Manyà (Eds.). Springer, 453–470. https://doi.org/10.1007/978-3-030-80223-3_31

Philipp Rümmer. 2008. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LNCS, Vol. 5330)*. Springer, 274–289.