An Eager Satisfiability Modulo Theories Solver for Algebraic Datatypes

Amar Shah, Federico Mora, Sanjit A. Seshia

University of California, Berkeley

amarshah1000@berkeley.edu, fmora@berkeley.edu, sseshia@eecs.berkeley.edu

Abstract

Algebraic data types (ADTs) are a construct classically found in functional programming languages that capture data structures like enumerated types, lists, and trees. In recent years, interest in ADTs has increased. For example, popular programming languages, like Python, have added support for ADTs. Automated reasoning about ADTs can be done using satisfiability modulo theories (SMT) solving, an extension of the Boolean satisfiability problem with first-order logic and associated background theories. Unfortunately, SMT solvers that support ADTs do not scale as state-of-the-art approaches all use variations of the same *lazy* approach. In this paper, we present an SMT solver that takes a fundamentally different approach, an *eager* approach. Specifically, our solver reduces ADT queries to a simpler logical theory, uninterpreted functions (UF), and then uses an existing solver on the reduced query. We prove the soundness and completeness of our approach and demonstrate that it outperforms the state of the art on existing benchmarks, as well as a new, more challenging benchmark set from the planning domain.

1 Introduction

Boolean satisfiability (SAT) solvers have been shown to efficiently solve a number of NP-hard problems in areas such as AI planning (Kautz, Selman et al. 1992), verification (Clarke et al. 2001), and software testing (Cadar et al. 2008). Satisfiability modulo theories (SMT) solvers are a natural extension to SAT solvers that can reason about first-order structures with background theories (Barrett et al. 2021), allowing them to tackle more general problems or to accept more succinct inputs. For example, SMT solvers can reason about bit-vectors (Brummayer and Biere 2009), floating-point numbers (Rümmer and Wahl 2010), strings (Bjørner et al. 2012), and algebraic data types (ADTs) (Barrett, Fontaine, and Tinelli 2017).

The power behind ADTs lies in how they can succinctly express complex structures at a high-level of abstraction while avoiding common programming pitfalls, like null pointer dereferencing (Hoare 1975). For most of their history, ADTs lived exclusively inside functional programming languages, like NPL (Burstall 1977), Standard ML (Milner 1997), and Haskell (Hudak et al. 2007). Recently, however, the interest in ADTs has exploded with a number of mainstream languages being released with support for ADTs, e.g., Rust (Jung et al. 2021), or have added support, e.g., Python (Salgado 2023) and Java (Goetz 2022).

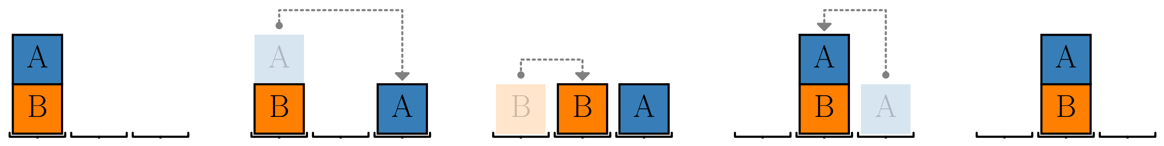
Automated reasoning about ADTs is important because this construct appears in many different software applications. As the popularity of ADTs grows, the demand for efficient SMT solvers will continue to increase. Unfortunately, the state-of-the-art tools in this space are already struggling to keep up. We demonstrate this empirically by generating a new benchmark set and showing that existing solvers, working together, are only able to solve 56.2% of the new queries in under 20 minutes per query (Sec. 5.2).

This imbalance between programming languages and SMT solvers is due to a gap in the SMT solving literature. Oppen (1980) was the first to give a decision procedure for the quantifier-free theory but ADTs do not seem to have permeated the community much further. In 2003, a consorted effort to unify the SMT community began with the first official input standard and competition, called SMT-LIB and SMT-COMP (Barrett et al. 2011), respectively. ADTs were not officially integrated into the standard until 2017, as part of version 2.6 (Barrett, Fontaine, and Tinelli 2017). In the most recent iteration of SMT-COMP, only two solvers participated in the ADT track, the least of any track, and both solvers use a variation of the same solving approach: a *lazy* SMT architecture combined with theory-specific reasoning based on the work by Oppen from 1980 (see Sec. 6).

We propose a new solving technique that departs from the standard approach in the community. Instead of a lazy approach, we take an *eager* approach (Barrett et al. 2021) that translates the original SMT formula into an equi-satisfiable formula without ADT elements. Our work fills the gap in the literature on SMT solving for ADT queries, and, by doing so, solves more queries than existing solvers (see Sec. 5.1). More importantly, we make the largest empirical contribution to the solving community on SMT-COMP benchmarks, solving different queries than existing tools (see Sec. 5.2).

1.1 Overview and Contributions

The rest of this paper is organized as follows. In Sec. 2 we describe ADTs, satisfiability, and our approach through an example planning problem called blocks world. In Sec. 3 we formally define ADTs and give the necessary background



(a) Initial configuration (b) First move (c) Second move (d) Third move (e) Target configuration

Figure 1: Solution (1b, 1c, and 1d) to a simple blocks world puzzle. 1a is the initial configuration; 1e is the target configuration.

on first-order logic and model theory to understand our approach. In Sec. 4 we describe our eager reduction from ADT queries to queries containing only uninterpreted functions (UF). Sec. 4 includes a proof of soundness and completeness along with a complexity analysis. In Sec. 5 we describe a prototype implementation of our approach, called Algaroba, and we evaluate it over two research questions. We find that Algaroba outperforms the state of the art overall and in terms of contribution to the solving community. Sec. 5 also describes a new benchmark set consisting of blocks world queries. This set addresses an important gap in the existing literature: the queries in this set contain all important kinds of ADTs, but are not easy to solve. We survey related work in Sec. 6 and then conclude in Sec. 7 with future work. Overall, we make the following contributions.

1. We define a notion of query *depth* and a finite, quantifier-free reduction from ADT queries to UF queries that uses depths. This is a new eager solving approach.
2. We prove the soundness and completeness of our approach and show that it generates at most a finite number of assertions.
3. We generate a new benchmark set that contains all important kinds of ADTs and is not trivial to solve. Existing benchmarks do not enjoy both these properties.
4. We implement our reduction approach in a prototype tool called Algaroba and we compare its performance to the state of the art. We find that Algaroba outperforms existing tools and that it makes the largest empirical contribution to the community of solvers.

2 Illustrative Example and New Benchmark

To better understand ADTs, satisfiability queries, our approach, and the benchmark set that we generate in Sec. 5, consider the classic blocks world problem first proposed by Winograd (1971). We use a version of the problem that Sussman (1973) used to illustrate the Sussman anomaly (Russell 2010) and that Gupta and Nau (1992) used to show that the associated decision problem is NP-Hard.

In the simplified version of the blocks world problem, there is a table with (possibly empty) stacks of blocks on it (an initial configuration), a desired way that the stacks should be arranged (a target configuration), and three rules:

1. blocks can only be taken from the top of a stack;
2. blocks can only be placed on the top of a stack; and
3. only one block can be moved at a time.

The general problem is to find a sequence of legal moves that leads from the initial configuration to the target configuration. The associated decision problem is to determine if there is such a sequence of length less than some input k.

Fig. 1 shows an example blocks world solution. The initial configuration is given in Fig. 1a, the target configuration in Fig. 1e, and Figs. 1b, 1c, and 1d show a sequence of three legal moves that solve the problem, where faded blocks denote the previous position of the block moved at that step.

The blocks world problem is a useful illustrative example for an ADTs solver because the encoding uses three important kinds of ADTs: sum, product, and inductive types. The following OCaml code gives the required type definitions for the example in Fig. 1.

```
1 type block = A | B
2 type tower =
3   | Empty
4   | Stack of {top: block; rest: tower}
5 type config =
6   | table of {l:tower; c:tower; r:tower}
```

Specifically, this code defines an enumerated type for blocks (`block` at line 1), a record type for table configurations (`config` at line 5), and an inductive type for stacks (`tower` at line 2). Variables of an enumerated type can take on any of the values listed in the type definition. For example, variables of type `block` can take on the values `A` or `B`. Variables of a record type can take on any combination of values of the type arguments listed in the type definition. For example, variables of type `config` can take on a triple of any three `tower` values. Enumerated types are the simplest form of a *sum* type, while records are the simplest form of a *product* type. ADTs allow for definitions that are both sum and product types. For example, variables of type `tower` can either be `Empty` or they can be a `Stack` but not both (sum). When these variables take on a `Stack` value, they are a pair of `block` and `tower` values (product). Notice that the definition of `tower` depends on itself. This makes `tower` an *inductive* type as well.

The blocks world problem is a useful illustrative example for satisfiability queries for two reasons. First, satisfiability-based solutions for similar planning problems have been around for decades (Kautz and Selman 1996). Second, encoding the problem as a satisfiability problem is simple when using bounded model checking (Biere et al. 1999) for bounded-horizon planning (e.g., see Rintanen (2003)). Specifically, the bounded model checking-based encoding is given by a transition system and a specification. The transition system starts at the initial configuration, and, at each step, makes a non-deterministic choice of which legal move to make. The specification is that the transition system never reaches the target configuration. Given this transition system, specification, and a bound k, bounded model checking will generate a satisfiability query whose solutions are assignments to the non-deterministic choices—concrete choices that get us to the target configuration from the initial configuration in less than k steps. SMT solvers, like our new solver, generate these solutions or prove that none exist. We use this encoding in Sec. 5 to generate a new benchmark set.

The blocks world problem also gives a useful intuition for our approach. While variables of inductive data types, like `tower`, can take on arbitrarily large values (e.g., a stack of a million `A` blocks), there is a bound on the size of relevant values. For the blocks world problem in Fig. 1 it would never make sense to have a `tower` value of size greater than two. Such a bound exists for all quantifier-free ADT queries; the problem is that automatically inferring the bound and using it to speed up solving was non-trivial. In this paper, we give an automated procedure for computing an over-approximation of this bound, and then we use this over-approximation to replace ADT functions with uninterpreted functions along with quantifier-free axioms.

3 Background

We assume a basic understanding of first-order logic. For a complete introduction, we refer the reader to Lubarsky (2008). Many-sorted first-order logic is like first-order logic but with the universe partitioned into different sorts (Barrett et al. 2021). We use many-sorted first-order logic and assume all terms are well-sorted, i.e. that we never apply a function to a term of the incorrect sort. In practice, standard type checking algorithms will catch these issues.

A *first-order theory* is a set of formulas (axioms). For example Equality and Uninterpreted Functions (Burch and Dill 1994) (**UF**) uses axioms to restrict the possible interpretations of $=$ (reflexivity, symmetry, transitivity) and all function symbols (congruence). A *satisfiability modulo theories (SMT) query* is a formula-theory pair. For a formula ϕ, there are free variables (which we will just call variables) and uninterpreted function symbols that we (usually) want to find an interpretation for. A *structure* is a universe along with a function that describes how all non-logical symbols must be interpreted over the universe. When a structure $\mathcal{M}$ satisfies a formula ϕ, then we say that $\mathcal{M}$ is a *model* of ϕ and we denote this as $\mathcal{M} \models \phi$. As a slight abuse of notation, for a set of formulas $\Phi = \{\phi_i\}$, we use $\mathcal{M} \models \Phi$ to mean $\mathcal{M} \models \bigwedge_1^n \phi_i$. Similarly, we use $\phi \models \psi$ to mean every model of ϕ is a model of ψ. For a model $\mathcal{M}$, we use the notation $\mathcal{M}[x]$ or $\mathcal{M}[f]$ to represent the variable x or the function f interpreted in the model $\mathcal{M}$. We say that an SMT query $(\phi, \mathbf{T})$ is **sat** iff there exists a model $\mathcal{M}$ such that $\mathcal{M} \models \mathbf{T} \cup \{\phi\}$. Otherwise we say the query is **unsat**. *SMT solvers* (Barrett et al. 2021) take an SMT query and return **sat** if there is an assignment to all functions and variables that satisfies the formula and the theory axioms. The distinctive aspect of SMT solvers is that they perform an encoding to SAT, either implicitly or explicitly. *Eager* SMT solvers perform a satisfiability-preserving reduction to SAT in a single phase (e.g., see Seshia (2005)) whereas *lazy* solvers perform an iterative encoding, on demand.

- $\forall \vec{s}\ is\text{-}f(f(\vec{s})) = \texttt{True}$
- $\forall \vec{r}\ is\text{-}f(g(\vec{r})) = \texttt{False}$ for constructors $g \neq f$
- $\forall \vec{t}\ f^i(f(\vec{s})) = \vec{s}_i$ for every selector f^i of f
- $\forall t\ is\text{-}f(t) \rightarrow \exists \vec{s}\ f(\vec{s}) = t$
- $\forall t \forall s$ if s is a descendant of t, then $s \neq t$

Figure 2: Axioms for corresponding constructors f, testers $is\text{-}f$, and selectors f^i. The last axiom is *acyclicality*.

A *theory literal* is a logical formula with no conjunctions ($\wedge$) or disjunctions ($\vee$). These are the base units of SMT solving and are the equivalent of *literals* in SAT solving. Our approach will be easier to understand when queries are in *negation normal form (NNF)* and *flat*. A formula is in NNF if only theory literals are negated. It is flat if all theory literals are of the form $\neg(x_1 = x_2)$, $x_1 = x_2$, $x_1 = g(x_2, ..., x_n)$ where x_i are variables. We will transform **ADT** queries into **UF** queries through a *theory reduction*.

Definition 3.1 (Theory Reduction)**.** *A theory **T** reduces to a theory **R** if there is a computable function m such that*

$$(\psi, \boldsymbol{T}) \textit{ is } \textbf{\textit{sat}} \leftrightarrow (m(\psi), \boldsymbol{R}) \textit{ is } \textbf{\textit{sat}}$$

3.1 Theory of Algebraic Data Types

We denote the theory of algebraic data types as **ADT**. It contains the full theory of **UF** and additional structure given by:

Definition 3.2 (**ADT** (Barrett, Fontaine, and Tinelli 2017))**.** *(1) An ADT $\mathcal{A}$ with sort σ is a tuple consisting of:*

- *A finite set of constructor functions $\mathcal{A}^C$, where we say a function $f : \sigma_1 \times ... \times \sigma_l \rightarrow \sigma$ has sort σ and arity l*
- *A finite set of selectors $\mathcal{A}^S$, such that there are m selectors $f^1, ..., f^m$ with $f^i : \sigma \rightarrow \sigma_i$ for each constructor $f \in \mathcal{A}^C$ with arity m.*
- *A finite set of testers $\mathcal{A}^T$ and a bijection $p : \mathcal{A}^C \rightarrow \mathcal{A}^T$ which sends $f \mapsto is\text{-}f$ where $is\text{-}f : \sigma \rightarrow \{\mathit{True}, \mathit{False}\}$*

(2) Every ADT $\mathcal{A}$ satisfies the axioms given in Fig. 2, where for two terms s and t, if s can be obtained by applying a sequence of l selectors to t, then we say s is an l^th descendant *of t and t is the l^th* ancestor *of s.*

An ADT term is any expression of an ADT sort. The set of *normal* ADT terms is the smallest set containing (1) constants (0-ary constructors), (2) constructors applied to only normal ADT terms. It is useful to think of normal terms as trees: constants are leaves and we can build larger trees by applying constructors to normal terms.

As an example, the `tower` definition from earlier uses two constructors: `Empty` and `Stack`. These are the two possible ways to build a `tower`. `Empty` is a function that takes no inputs and outputs a `tower`. `Stack` is a

Algorithm 1: Reduce(ψ)

```
$\psi_1 \leftarrow NNF(\psi)$
$\psi_2 \leftarrow Flatten(\psi_1)$
$k \leftarrow$ Number of ADT variables in $\psi_2$
$\psi_3 \leftarrow$ Apply Fig. 4a rewrite rules A & B to $\psi_2$
$\phi_1, ..., \phi_m \leftarrow$ Apply Fig 4b 1, 2, & 3 using $k$ to $\psi_3$
$\psi^* \leftarrow \psi_3 \wedge \phi_1 \wedge ... \wedge \phi_m$
return $UF\text{-}SMT\text{-}Solver(\psi^*)$
```

function that takes a `block` and a `tower` and outputs a `tower`. Each corresponding constructor has a set of selectors. `Empty` has no selectors and so it is a normal term, but `Stack` has two selectors, `top` and `rest`. In OCaml, we apply selectors using dot notation, e.g., `x.top`. Selectors can be thought of as de-constructors—we use them to get back the terms constructed a `tower`. The `tower` definition implicitly defines two testers: `is-Empty` and `is-Stack` These predicates take a `tower` and return true iff the argument was built using the matching constructor.

4 Eager Reduction of ADT to UF

We propose a new SMT solver for the **ADT** theory. For a quantifier-free input formula ψ, our solver generates a quantifier-free formula ψ^* in **UF** and then calls an existing **UF** solver to get a final result. We cannot compute a quantifier-free reduction directly, since **ADT** axioms have universal quantifiers. Instead, we only instantiate the **ADT** axioms over terms that are relevant to the input query.

When the universe of the input query is finite (e.g., it only contains enums), we instantiate the entire universe (see Sec. 4.1). Otherwise, we follow the procedure in Alg. 1. This procedure transforms the input query to NNF, flattens the result, and then applies the rewrite rules in Fig. 4a and adds the axioms Fig. 4b. The *depth* of a query is the number of variables in the flattened NNF version of the query. In Alg. 1, the depth is k. The depth is linear in the size of the input query because the NNF and flattening transformations introduce at most a linear number of variables.

Rules A and B from Fig. 4a correspond to rewriting constructor and selector applications respectively so that they work well with other constructor, selectors and testers. Rule B contains existential quantifiers, but these are handled through *Skolemization* (replacing existentially bound variables by free variables). Axioms 1 and 2 from Fig. 4b ensure that only one tester returns true for each term, and that the this tester corresponds to a constant iff the term is a constant. Axiom 3 encodes the **ADT** acyclicality constraint.

To better understand Axiom 3 and acyclicality, consider the following example query. Let `x` and `y` be of type `tower` defined in Sec. 2. The query

```
(is-Stack x) && (is-Stack y)
  && (y = x.rest) && (x = y.rest)
```

is **unsat** because any satisfying assignment would need to violate acyclicality. Fig. 3 illustrates this: there is a circular dependency between `x` and `y`. Therefore, to avoid spurious models, we must encode the acyclicality property in our reduction. The challenge is that we need to capture this seemingly infinite property using a only finite number of quantifier-free formulas.

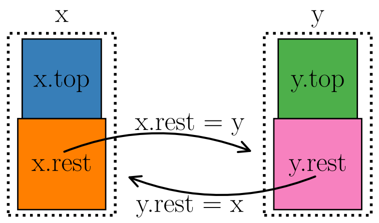


Figure 3: Visual representation of an **unsat** query.

Our key result is that we only need to enforce acyclicality for all $l < k^{\text{th}}$ descendants, where k is the number of ADT variables in the flattened query. To see the intuition behind this, consider the following generalization of the previous example. Let x_1, ..., x_k be of type `tower`. The flat query

```
(is-Stack $x_1$) &&...&& (is-Stack $x_k$) &&
($x_2$ = $x_1$.rest) && ($x_3$ = $x_2$.rest) &&...&&
($x_k$ = $x_{k-1}$.rest) && ($x_1$ = $x_k$.rest)
```

asserts that there is a cycle of size k. Since, our reduction asserts acyclicality for all $l < k^{\text{th}}$ descendants, we correctly return unsat on this query. Furthermore, it is impossible to have a query with a cycle of size more than k using k or fewer variables (in the flat query), so our encoding is sufficient (see Sec. 4.3 for a full proof).

4.1 Finite Universe Instantiation

If we recognize an ADT has a finite universe, we create constants for every term in the universe and instantiate the axioms over the entire, finite universe. This is a source of double exponential blowup, but ADTs with finite universes are rare and often small enough to prevent noticeable blowup.

4.2 Complexity Analysis

In the finite universe case, we can have a doubly exponential blowup. One adversarial case is an ADT that is records of records of enums:

```
type enum = A | B
type rec1 = j of {l: enum; r: enum}
type rec2 = k of {m: rec1; s: rec1}
```

Here, `enum` has a universe of size two, `rec1` has a universe of size four, and `rec2` has a universe of size 16. This gives a double exponential blowup since we are creating variables to represent every normal term of every datatype.

In the infinite universe case, we have at worst an exponential blowup in the size of the query. We know the depth k is at most linear in the size of the query, however, for a term x of the `tree` type definition below, the number of sequences of selector applications of length up to k is $2^{k+1} - 2$, thus giving us an exponential blowup in the number of terms.

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

4.3 Proof of Correctness

In this proof when we refer to a rule or axiom, we mean those from Fig. 4a and Fig. 4b, respectively. We assume ψ is

A. $f(\vec{s}) = t \Longrightarrow f(\vec{s}) = t \wedge \textit{is-}f(t) \wedge \bigwedge_{i=1}^{m} f^i(t) = \vec{s}_i$
B. $f^j(t) = t_j \Longrightarrow f^j(t) = t_j \wedge$
$[\textit{is-}f(t) \rightarrow [\exists \vec{s}[f(\vec{s}) = t \wedge \bigwedge_{i=1}^{m} f^i(t) = \vec{s}_i]]]$

(a) Rewrites

1. Add $\bigvee_{i=1}^{|\mathcal{A}_T|}[\textit{is-}f_i(t) \wedge \bigwedge_{j=1, j\neq i}^{|\mathcal{A}_T|} \neg \textit{is-}f_j(t)]$
2. For any constant constructor c, add $\textit{is-}c(t) \leftrightarrow c = t$
3. If s is the l^{th} descendant of t and $l < k$, add $s \neq t$

(b) Axioms

Figure 4: Rewrite rules (a) and additional axioms (b) used in Alg. 1

flattened and in NNF (i.e., it is ψ_2 in Alg. 1). For simplicity, we also assume that ψ does not contain any uninterpreted functions or sorts, but it is not difficult to see how to extend the proof to include them.

Theorem 4.1. *Alg. 1 shows that **ADT** reduces to **UF**. Specifically, $(\psi, \textbf{ADT})$ is **sat** $\leftrightarrow$ $(\psi^*, \textbf{UF})$ is **sat**.*

Proof. $\underline{\rightarrow}$: If $(\psi, \textbf{ADT})$ is **sat**, then there is a model $\mathcal{M} \models \psi \cup \textbf{ADT}$. ψ^* is ψ modified according to rules A and B and axioms 1, 2, and 3. Each of these rules and axioms is consistent with the axioms of **ADT** in Definition 3.2 and thus $\mathcal{M} \models \textbf{ADT} \cup \psi^*$. Since every model in **ADT** is a model in **UF**, $\mathcal{M} \models \textbf{UF} \cup \psi^*$. Thus, $(\psi^*, \textbf{UF})$ is **sat**

$\underline{\leftarrow}$: If $(\psi^*, \textbf{UF})$ is **sat**, then we know that there is some model $\mathcal{N} \models \textbf{UF} \cup \psi^*$. We will assume that ψ^* is a conjunction of theory literals. This is permissible since the modification from ψ to ψ^* involves either replacing a theory literal with a conjunction of theory literals or adding on conjunctions of theory literals to the end of the formula. Thus, the satisfying assignment to the propositional structure of ψ^*, will be a superset of a satisfying assignment to the propositional structure of ψ.

We want to modify $\mathcal{N}$ to create a model $\mathcal{M} \models \textbf{ADT} \cup \psi$. As we describe in Section 4.1, if the universe is finite, we manually instantiate all of the ADT normal terms in the query. Thus, in the finite universe case, it must be that $\textbf{ADT} \models \psi$. We now assume the **ADT** universe is infinite.

Since we want $\mathcal{M} \models \textbf{ADT}$, we set the universe of $\mathcal{M}$ to be all of the ADT normal terms. Consider the set of variables that appear in ψ^* which we call $V = \{x_1, ..., x_k\}$. We describe an algorithm that for all $x \in V$, sets $\mathcal{M}[x]$ to some ADT normal term, such that we ultimately get $\mathcal{M} \models \psi$.

V is the set of variables in ψ^*, thus for each $x \in V$, by axiom 1 there is exactly one tester *is-f* such that $\mathcal{N} \models \textit{is-f}(x)$.

There are two "base cases" for our construction of $\mathcal{M}$. First, if f is some constant constructor, by axiom 2 of the reduction, we know that $\mathcal{N}[x] = \mathcal{N}[f]$, so we set $\mathcal{M}[x] \triangleq \mathcal{M}[f]$. Second, if x is some variable that is never set equal to some constructor application or selected from (either directly or transitively) then we set $\mathcal{M}[x]$ to an ADT normal term. Since our **ADT** universe is infinite, we will specifically pick an ADT normal term t such that it takes at least $k + 1$ selector applications to get to any of the ADT normal terms that we have already set. This will prevent any of our different ADT normal term assignments from interfering with each other—they are too far away in the infinite universe.

If we are not in one of these base cases, we know that f is an m-ary constructor for some $m > 0$. Since x was either constructed or selected, there are variables $y_1, ..., y_m$ in V such that $\mathcal{N} \models \bigwedge_{i=1}^{m} f^i(x) = y_i$. Note that these variables are from the original query if x is equal to a constructor application, or from Skolemization if x is selected from.

Continuing our construction of $\mathcal{M}$, we recurse on these y_i that have not already been assigned in $\mathcal{M}$. We will eventually hit a base case, since there are a finite number of selector/constructor applications in our original query. For each i, we set $\mathcal{M}[f^i](\mathcal{M}[x]) \triangleq \mathcal{M}[y_i]$. Finally, we set $\mathcal{M}[x] \triangleq \mathcal{M}[f](\mathcal{M}[y_1], ..., \mathcal{M}[y_m])$.

If it were possible to have

$$\psi \models f(y_1, ..., y_m) \neq x \tag{1}$$

hold, then we would have $\mathcal{M} \not\models \psi$ and our current proof attempt would not go through. However, we will show that this is never the case.

Note that since $\mathcal{N} \models \psi^*$, if ψ^* asserts anything about selector applications, these selector applications must be consistent with $\mathcal{N}$. Also, ψ^* must assert something about selector applications, since we know that x is either equal to a constructor application or is selected from in the query. Thus, $\psi^* \models \bigwedge_{i=1}^{m} f^i(x) = y_i$, meaning that ψ^* asserts the correct selector behavior. We now use this to guarantee the correct constructor behavior.

There are two ways that incorrect constructor behavior could occur in ψ^*: 1. If $\psi \models f(y_1, ..., y_m) = x$ which contradicts Equation (1). 2. If $\psi \models \bigwedge_{i=1}^{m} y_i = f^i(x)$, but then by rule B, we would still have $\psi^* \models f(y_1, ..., y_m) = x$ which also contradicts Equation (1) since $\psi^* \models \psi$.

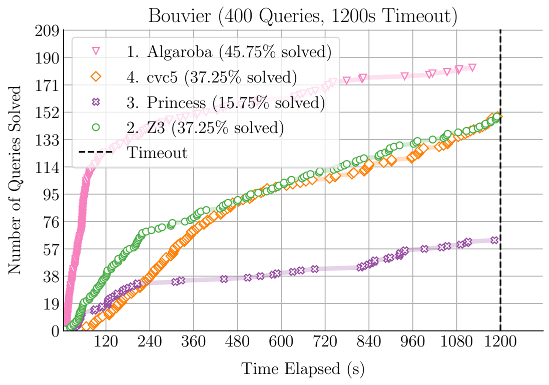
We iterate this construction of $\mathcal{M}$ for each variable in V for at most k rounds, since there are k total variables in V. Thus, since ψ^* has the acyclicality Axiom 3 instantiated up to a depth k, we do not create any cycles in $\mathcal{M}$.

We can also see that $\mathcal{M} \models \psi$ since each theory literal ψ_i in $\psi = \bigwedge_{i=1}^{p} \psi_i$ will be an equality $x = y$, disequality $x \neq y$, selector application $f^j(y) = x$, a tester application *is-f*(x), or a constructor application $f(x_1, ..., x_m) = y$. If it is any of these, then $\mathcal{M} \models \psi_i$ by how we defined $\mathcal{M}$. Note that if it was a constructor application, then by rule A ψ^* would have the respective selector applications and thus our construction of $\mathcal{M}$ would satisfy ψ_i.

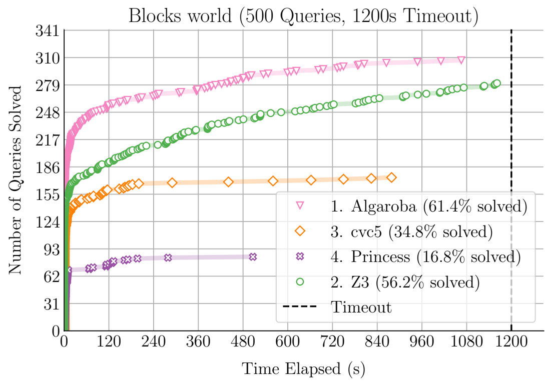
Thus, $\mathcal{M} \models \textbf{ADT} \cup \psi$ and so $(\psi, \textbf{ADT})$ is **sat**. □

5 Empirical Evaluation

In this section we empirically compare the performance of our approach to state-of-the-art solvers. Specifically, we aim



(a) Bouvier benchmark set.



(b) Blocks world benchmark set.

Figure 5: 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.

to answer the following research questions.

RQ1 How does the overall performance of our approach compare to the state of the art?

RQ2 How complementary is the performance of our approach to that of existing solvers?

We implement a prototype of our approach, called Algaroba,[1] in approximately 2900 lines of OCaml code. We use the Z3 API as the default **UF** back-end solver but we allow for any **UF** solver to be used instead. 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. The state-of-the-art solvers in this space are cvc5 (we use version 1.0.6-dev.214.97a64fc16) and Z3 (we use version 4.12.2). We also include Princess (latest release as of 2023-06-19) in our evaluation since it is the most related approach. We describe all three solvers in Sec. 6.

Our evaluation covers two existing benchmark sets from SMT-COMP, one originally from Bouvier (2021) and one originally from Barrett, Shikanian, and Tinelli (2007) (BST for short). These two benchmark sets are useful but limited: every solver succeeds on every BST query so it is difficult to draw performance conclusions; Bouvier queries are more challenging but only contain sum types.

To address these limitations, we introduce a new benchmark set consisting of randomly generated blocks world queries.[1] Blocks world queries, which we describe in Sec. 2, are more challenging to solve than those in BST and, unlike those from Bouvier, contain sum, product, and inductive types. To generate blocks world queries we use the same table configuration as in Sec. 2 (three places for towers), but we randomly select a set of blocks (ranging between two to 26) and we randomly generate an initial and target configuration (two sets of three random block towers). We call these three random samples a blocks world setup. For each blocks world setup, we randomly sample a set of step numbers (ranging from one to two times the number of blocks) and generate a blocks world query for each step number. This process resulted in 500 individual queries that each ask "can we get from this initial configuration to this target configuration in exactly this number of steps?"

[1]available at https://github.com/uclid-org/algaroba/tree/aaai24

5.1 RQ1: Overall Performance

To answer our first research question, we time the execution of Algaroba, cvc5, Princess, and Z3 on all queries in all three benchmark sets. When more than one solver terminates on a given query we compare the results to check for disagreements. There was not a single case where one solver returned **sat** and another returned **unsat**; therefore, we focus the remainder of our evaluation on execution times.

For the BST benchmark set, which consists of 8000 queries, every solver successfully terminates on every query within the timeout. cvc5 performs the best on average with an average solve time of 0.05 seconds (compared to 0.08 seconds for Algaroba and 0.10 seconds for Z3). Z3 performed the most consistently with a standard deviation of 0.05 seconds (compared to 0.10 seconds for Algaroba and 0.15 seconds for cvc5). Given the magnitude of these values, we conclude that the performance differences between Algaroba, cvc5, and Z3 are negligible on this set. Princess is the slowest (2.20 seconds on average) and least consistent (standard deviation of 1.69 seconds) but still effective.

Results are more interesting for the remaining benchmark sets. Fig. 5a shows the execution times for every solver on every query in the Bouvier benchmark set (excluding timeouts). No solver succeeds on more than half the queries in the set but Algaroba clearly outperforms the rest. In terms of number of queries solved, Algaroba succeeds on 8.5 percentage points more queries than the next best (45.75% versus cvc5 and Z3's 37.25%). In terms of average run time on successful queries, Algaroba is 2.30 times faster than the next best (190.11 seconds versus Z3's 437.18 seconds). In terms of standard deviation on successful queries, Algaroba is 1.30 times more consistent than the next best (267.13 seconds versus cvc5's 346.83 seconds).

Fig. 5b shows the execution times for every solver on every query in the blocks world benchmark set (excluding timeouts). Again, Algaroba outperforms the state-of-the-art solvers. Algaroba solves 5.2 percentage points more queries than the next best (61.4% versus Z3's 56.2%) but the average and standard deviation results are more complicated. Princess is the fastest and most consistent solver on solved queries (31.89 seconds and 74.77 seconds, respectively) but it succeeds on 44.6 percentage points fewer queries than Algaroba. Compared with Z3, which solves the second most number of queries, Algaroba is 2.00 times faster (87.34 seconds on average compared to Z3's 175.05 seconds) and 1.51 times more consistent in terms of standard deviation (198.67 seconds versus Z3's 300.15 seconds).

Across both interesting benchmarks, Algaroba solves the most **sat** queries (167 versus cvc5's 70), and the second most **unsat** queries (322 versus Z3's 406). The average (median) increase in query size from our reduction was 259x (146x). Given the overall success of Algaroba on both interesting benchmark sets, we answer RQ1 by concluding that our performance compares favorably to the state of the art.

5.2 RQ2: Contribution Rank

Measuring overall performance is useful but it does not give an accurate perspective on how the community uses these tools. When faced with an SMT query, practitioners are likely to use multiple different solvers. This could be in parallel, like (Rungta 2022), or through algorithm selection, like (Pimpalkhare et al. 2021). *Contribution ranks* capture this practical perspective by evaluating solvers in terms of how complementary they are to other solvers. A higher rank means a higher contribution to the community of solvers.

To evaluate how complementary our approach is to existing solvers, we use SMT-COMP's contribution ranking. This ranking uses the notion of a *virtual best* solver, which is defined as $vb(q, S) \triangleq s(q)$, where S is a set of solvers and s is the solver in S that terminates most quickly on q. Informally, the ranking answers, "which solver can I remove from the virtual best solver to hurt performance the most?"

In terms of number of queries solved (the primary SMT-COMP metric), there is a four-way tie on the BST benchmark set—all solvers solve all queries. For both other benchmark sets, Algaroba is ranked highest. For blocks world, the virtual best solver without Algaroba succeeds on 56.2% of the queries, less than Algaroba on its own (61.4%). With Algaroba, the virtual best solver succeeds on 62.2%. For Bouvier, without Algaroba, the virtual best solver succeeds on 64.25% of the queries. With Algaroba, this number rises to 83.75%. These positive results are in part because Algaroba solves the most queries, but are mainly due to the uniqueness of our approach. cvc5 and Z3 use a similar underlying algorithm, so removing one does not affect the performance of the virtual best solver. On the other hand, while Princess is the most similar approach to our own, their reduction is different enough to not interfere with our ranking. In short, we solve many queries that no other solver can (108/900).

Given the winning contribution rank of Algaroba on both interesting benchmark sets, we answer RQ2 by concluding that our performance is complementary to existing solvers—we solve many benchmarks that no other solver can.

6 Related Work

Most solvers for quantifier-free ADT queries use a lazy SMT architecture, i.e., they use a theory specific solver to handle the data types and a core solver to handle the logical formula (Sebastiani 2007). A common theory solver will use a combination of congruence closure, syntactic unification, and acyclicality checks (Barrett, Shikanian, and Tinelli 2007; Oppen 1980; Reynolds and Blanchette 2017; Reynolds et al. 2018). This is the case for popular SMT solvers like cvc5 (Barbosa et al. 2022), SMTInterpol (Christ, Hoenicke, and Nutz 2012), and Z3 (de Moura and Bjørner 2008). cvc5 and SMTInterpol were the only two participants in the most recent SMT-COMP for quantifier-free ADT queries. We differ in that we take an eager approach.

Princess (Hojjat and Rümmer 2017) also takes an eager approach. However, Princess reduces queries to **UF** and Linear Integer Arithmetic (**LIA**). **LIA** makes keeping track of the depth of ADT terms easy, but their reduction results in queries that are more difficult to solve (see Sec. 5).

The scope of our work is quantifier-free ADT queries. However, there is existing related work that deals with quantifiers. De Angelis et al. (2020) and Kostyukov, Mordvinov, and Fedyukovich (2021) provide approaches to solving ADT Constrained Horn Clauses (CHCs). Other approaches (Suter, Dotta, and Kuncak 2010; Pham and Whalen 2014) support restricted forms of recursive functions (called catamorphisms) via partially evaluating these functions. Kovács, Robillard, and Voronkov (2017) provides two decision procedures for quantified ADTs.

7 Conclusions

As the popularity of ADTs continues to grow, the demand for efficient SMT solvers that can handle ADTs will increase. Unfortunately, there are few existing solvers in this space and the performance of these solvers can be improved.

We introduced a reduction from quantifier-free **ADT** queries to quantifier-free **UF** queries. This approach is sound, complete, and eager, while most existing approaches are lazy. We implemented a prototype tool of our approach and compared with against existing solvers. We found that we can solve more queries using less time. More importantly, we found that we make the largest empirical contribution to the solving community.

In the future, we intend to support proof generation, quantifiers, and hybrid eager and lazy approaches. We will also experiment with different back-end solvers and techniques for automatically selecting back-ends per input query.

Acknowledgements

We would like to thank Adwait Godbole, Ameesh Shah, Jiwon Park and Shangyin Tan for their insightful feedback. This work was supported in part by a Qualcomm Innovation Fellowship, NSF grant 1837132, DARPA contract FA8750-20-C-0156, an Amazon Research Award, Toyota under the iCyPhy center, a UC Berkeley Summer Undergraduate Research Fellowship, and by Intel under the Scalable Assurance program.

References

Barbosa, H.; Barrett, C.; Brain, M.; Kremer, G.; Lachnitt, H.; Mann, M.; Mohamed, A.; Mohamed, M.; Niemetz, A.; Nötzli, A.; Ozdemir, A.; Preiner, M.; Reynolds, A.; Sheng, Y.; Tinelli, C.; and Zohar, Y. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In Fisman, D.; and Rosu, G., eds., *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 415–442. Cham: Springer International Publishing.

Barrett, C.; de Moura, L.; Ranise, S.; Stump, A.; and Tinelli, C. 2011. The SMT-LIB Initiative and the Rise of SMT: (HVC 2010 Award Talk). In *Hardware and Software: Verification and Testing*, 3–3. Springer.

Barrett, C.; Fontaine, P.; and Tinelli, C. 2017. The SMT-LIB Standard Version 2.6. https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf.

Barrett, C.; Sebastiani, R.; Seshia, S. A.; and Tinelli, C. 2021. Satisfiability Modulo Theories. In Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds., *Handbook of Satisfiability*, chapter 33, 1267–1329. IOS Press, second edition.

Barrett, C.; Shikanian, I.; and Tinelli, C. 2007. An Abstract Decision Procedure for Satisfiability in the Theory of Recursive Data Types. *Electronic Notes in Theoretical Computer Science*, 174(8): 23–37. Combined Proceedings of the Fourth Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR 2006) and the First International Workshop on Probabilistic Automata and Logics (PaUL 2006).

Biere, A.; Cimatti, A.; Clarke, E.; and Zhu, Y. 1999. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 193–207. Springer.

Bjørner, N.; Ganesh, V.; Michel, R.; and Veanes, M. 2012. An SMT-LIB format for sequences and regular expressions. *SMT*, 12: 76–86.

Bouvier, P. 2021. The VLSAT-3 Benchmark Suite. *INRIA Technical Report 516*.

Brummayer, R.; and Biere, A. 2009. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In Kowalewski, S.; and Philippou, A., eds., *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 174–177. Berlin, Heidelberg: Springer Berlin Heidelberg.

Burch, J. R.; and Dill, D. L. 1994. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification (CAV)*, 68–80. Springer.

Burstall, R. M. 1977. Design considerations for a functional programming language. *Proc. Infotech State of the Art Conf. "The Software Revolution"*, 45–57.

Cadar, C.; Ganesh, V.; Pawlowski, P. M.; Dill, D. L.; and Engler, D. R. 2008. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2): 1–38.

Christ, J.; Hoenicke, J.; and Nutz, A. 2012. SMTInterpol: An Interpolating SMT Solver. In Donaldson, A.; and Parker, D., eds., *Model Checking Software*, 248–254. Berlin, Heidelberg: Springer Berlin Heidelberg.

Clarke, E.; Biere, A.; Raimi, R.; and Zhu, Y. 2001. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19: 7–34.

De Angelis, E.; Fioravanti, F.; Pettorossi, A.; and Proietti, M. 2020. Removing Algebraic Data Types from Constrained Horn Clauses Using Difference Predicates. In Peltier, N.; and Sofronie-Stokkermans, V., eds., *Automated Reasoning*, 83–102. Cham: Springer International Publishing.

de Moura, L.; and Bjørner, N. 2008. Z3: An Efficient SMT Solver. In Ramakrishnan, C. R.; and Rehof, J., eds., *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 337–340. Berlin, Heidelberg: Springer Berlin Heidelberg.

Ershov, A. P. 1958. On Programming of Arithmetic Operations. *Commun. ACM*, 1(8): 3–6.

Goetz, B. 2022. JEP 360: Sealed Classes (Preview). https://openjdk.org/jeps/360. Accessed: 2023-08-15.

Gupta, N.; and Nau, D. S. 1992. On the complexity of blocks-world planning. *Artificial intelligence*, 56(2-3): 223–254.

Hoare, C. A. R. 1975. Recursive data structures. *International Journal of Computer & Information Sciences*, 4(2): 105–132.

Hojjat, H.; and Rümmer, P. 2017. Deciding and Interpolating Algebraic Data Types by Reduction. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 145–152.

Hudak, P.; Hughes, J.; Peyton Jones, S.; and Wadler, P. 2007. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 12–1.

Jung, R.; Jourdan, J.-H.; Krebbers, R.; and Dreyer, D. 2021. Safe systems programming in Rust. *Communications of the ACM*, 64(4): 144–152.

Kautz, H.; and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the national conference on artificial intelligence*, 1194–1201.

Kautz, H. A.; Selman, B.; et al. 1992. Planning as Satisfiability. In *ECAI*, volume 92, 359–363. Citeseer.

Kostyukov, Y.; Mordvinov, D.; and Fedyukovich, G. 2021. Beyond the Elementary Representations of Program Invariants over Algebraic Data Types. In *Programming Language Design and Implementation*, PLDI 2021, 451–465.

New York, NY, USA: Association for Computing Machinery. ISBN 9781450383912.

Kovács, L.; Robillard, S.; and Voronkov, A. 2017. Coming to Terms with Quantified Reasoning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, 260–270. New York, NY, USA: Association for Computing Machinery.

Lubarsky, R. 2008. Ian Chiswell and Wilfrid Hodges. Mathematical logic. Oxford Texts in Logic, vol. 3. Oxford University Press, Oxford, England, 2007, 250 pp. *Bulletin of Symbolic Logic*, 14(2): 265–267.

Milner, R. 1997. *The definition of standard ML: revised*. MIT press.

Oppen, D. C. 1980. Reasoning About Recursively Defined Data Structures. *J. ACM*, 27(3): 403–411.

Pham, T.-H.; and Whalen, M. W. 2014. An Improved Unrolling-Based Decision Procedure for Algebraic Data Types. In Cohen, E.; and Rybalchenko, A., eds., *Verified Software: Theories, Tools, Experiments*, 129–148. Berlin, Heidelberg: Springer Berlin Heidelberg.

Pimpalkhare, N.; Mora, F.; Polgreen, E.; and Seshia, S. A. 2021. MedleySolver: Online SMT Algorithm Selection. In Li, C.; and Manyà, F., eds., *Theory and Applications of Satisfiability Testing*, volume 12831 of *Lecture Notes in Computer Science*, 453–470. Springer.

Reynolds, A.; and Blanchette, J. C. 2017. A Decision Procedure for (Co)datatypes in SMT Solvers. *Journal of Automated Reasoning*, 58(3): 341–362.

Reynolds, A.; Viswanathan, A.; Barbosa, H.; Tinelli, C.; and Barrett, C. 2018. Datatypes with Shared Selectors. In Galmiche, D.; Schulz, S.; and Sebastiani, R., eds., *Automated Reasoning*, 591–608. Cham: Springer International Publishing.

Rintanen, J. 2003. Symmetry Reduction for SAT Representations of Transition Systems. In *ICAPS*, 32–41.

Rümmer, P.; and Wahl, T. 2010. An SMT-LIB theory of binary floating-point arithmetic. In *International Workshop on Satisfiability Modulo Theories (SMT)*, 151.

Rungta, N. 2022. A billion SMT queries a day. In *Computer Aided Verification (CAV)*, 3–18. Springer.

Russell, S. J. 2010. *Artificial intelligence a modern approach*. Pearson Education, Inc.

Salgado, P. G. 2023. What's New In Python 3.10. https://docs.python.org/3.10/whatsnew/3.10.html\#summary-release-highlights. Accessed: 2023-08-15.

Sebastiani, R. 2007. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3: 141–224.

Seshia, S. A. 2005. *Adaptive Eager Boolean Encoding for Arithmetic Reasoning in Verification*. Ph.D. thesis, Carnegie Mellon University.

Sussman, G. J. 1973. A Computational Model of Skill Acquisition. Technical report, Massachusetts Institute of Technology, USA.

Suter, P.; Dotta, M.; and Kuncak, V. 2010. Decision Procedures for Algebraic Data Types with Abstractions. *SIGPLAN Not.*, 45(1): 199–210.

Winograd, T. 1971. *Procedures as a representation for data in a computer program for understanding natural language*. AI-TR. M.I.T. Project MAC.