

Deciding and Interpolating Algebraic Data Types by Reduction (Technical Report)

Hossein Hojjat

Department of Computer Science
Rochester Institute of Technology, Rochester, NY, United States

Philipp Rümmer

Department of Information Technology
Uppsala University, Uppsala, Sweden

Abstract—Recursive algebraic data types (term algebras, ADTs) are one of the most well-studied theories in logic, and find application in contexts including functional programming, modelling languages, proof assistants, and verification. At this point, several state-of-the-art theorem provers and SMT solvers include tailor-made decision procedures for ADTs, and version 2.6 of the SMT-LIB standard includes support for ADTs. We study an extremely simple approach to decide satisfiability of ADT constraints, the reduction of ADT constraints to equisatisfiable constraints over uninterpreted functions (EUF) and linear integer arithmetic (LIA). We show that the reduction approach gives rise to both decision and Craig interpolation procedures in (extensions of) ADTs.

Keywords—Decision procedures; Craig interpolation; algebraic data types; term algebras

I. INTRODUCTION

Recursive algebraic data types (ADTs) with absolutely free constructors are increasingly supported by SMT solvers, and find application in a variety of areas, including functional programming, modelling languages, proof assistants, and verification. In solvers, ADTs are usually implemented as native theory solvers [2], [12], [13], [17] that apply congruence closure (upward closure), syntactic unification (downward closure), cycle detection (occurs-check), and in additional handle selectors and testers in order to decide satisfiability of quantifier-free ADT formulas.

In this paper, we study a simple alternative approach to ADT reasoning, based on the *reduction* of ADT formulas to equisatisfiable formulas over uninterpreted functions and linear integer arithmetic (EUF+LIA). Our approach is partly inspired, besides by eager SMT in general, by the reduction approach from [10], in which quantifier-free formulas are mapped to simpler theories for the purpose of checking satisfiability and computing interpolants. For instance, as shown in [10], the theory of sets with finite cardinality constraints can be reduced to the theory of equality with uninterpreted functions (EUF). Like in [10], the target theories of our ADT reduction are EUF and linear arithmetic. Unlike [10], we are able to completely avoid universal quantifiers in the process of reduction, but the reduction depends on the introduction of further uninterpreted functions (which create some additional work in interpolation, see Section V).

The main idea of reduction is to augment an ADT formula with additional literals that ensure that constructors, selectors, and testers are interpreted consistently, and

that constructors are free. EUF takes care of upward and downward closure, while cycle detection and constructor testers are handled by LIA constraints. The reduction can be implemented with little effort, and is widely applicable since EUF and LIA are supported by virtually all SMT solvers, and increasingly also by other theorem provers. Reduction to EUF+LIA has a few further advantages, in particular it is possible to reuse existing, highly optimised EUF+LIA simplifiers in solvers, and to compute interpolants using EUF+LIA interpolation procedures.

The contributions of the paper are (i) definition and correctness proof of the reduction from ADTs to EUF+LIA; (ii) discussion of Craig interpolation for ADTs; (iii) extension to ADTs with size constraints, and an effective characterisation of the ADTs for which the resulting procedure is complete. The procedures discussed in the paper have been implemented in the PRINCESS theorem prover [14].¹

A. Related Work

ADT Solving: While ADTs have only recently been standardised in the SMT-LIB, some solvers (including STeP [11], CVC3 [3], CVC4 [1], and Z3 [7]) have for a while supported ADTs through native decision procedures extending the congruence closure algorithm [2], [12], [13]. Native solvers offer excellent performance, but also require significant implementation effort. The listed solvers do not support Craig interpolation or formulas with size constraints.

Satisfiability of ADT formulas can also be checked by introducing explicit axioms about the constructors and selectors. Since ADTs form a local theory [16], the set of required instances of the axioms can effectively be computed, and a decision procedure for ADT satisfiability is obtained.

Our reduction-based approach sits in between native solvers and methods based on explicit axioms. Like with explicit axioms, our method leaves most of the heavy work to other theory solvers (EUF and LIA), and is therefore easy to implement. The reduction approach is structure-preserving, however, which makes us believe that it can utilise existing contextual simplifiers (pre-processors or in-processors) more effectively than approaches based on axioms; it also directly gives rise to an interpolation procedure.

¹<http://www.philipp.ruemmer.org/princess.shtml>

ADT Interpolation: It has been observed in [10] that the theory of ADTs has the interpolation property; this result directly follows from admissibility of quantifier elimination in ADTs [12]. To the best of our knowledge, our ADT solver implemented in PRINCESS is the *first proof-based interpolation procedure for ADTs*.

ADTs with Size Constraints: Our approach for handling ADT formulas with size constraints is inspired by the more general unfolding-based decision procedure for ADTs with abstractions (i.e., catamorphisms) in [17]. The algorithm in [17] is complete for *sufficiently surjective* abstraction functions, which includes the size function on binary trees, but *not* the size function on ADTs in general. We augment the setting from [17] by giving a necessary and sufficient criterion for sufficient surjectivity of the size function, and thus for completeness of the overall procedure.

ADTs with size constraints can also be represented in the local theory framework [16], again by introducing the necessary instances of explicit axioms.

A further decision procedure for ADTs with size constraints, based on the concept of length constraint completion, has been described in [19]. Our method uses the simple approach of *unfolding* in order to add size constraints to the overall reduction-based procedure; it is at this point unclear whether length constraint completion could be combined with the reduction approach as well.

II. PRELIMINARIES

We formulate our approach in the setting of multi-sorted first-order logic. The signature Σ of an ADT is defined by a sequence $\sigma_1^d, \dots, \sigma_k^d$ of sorts and a sequence f_1, \dots, f_m of constructors. The type $\alpha(f_i)$ of an n -ary constructor is an $(n+1)$ -tuple $\langle \sigma_0, \dots, \sigma_n \rangle \in \{\sigma_1^d, \dots, \sigma_k^d\}^{n+1}$, normally written in the form $f_i : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma_0$. Zero-ary constructors are also called constants. By slight abuse of notation, we also write $f_i : \sigma_j^d$ if the result type of f_i is σ_j^d , i.e., if $f_i : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma_j^d$ for some $\sigma_1, \dots, \sigma_n$.

In addition to constructors, formulas over ADTs can be formulated in terms of *variables* $x \in \mathcal{X}$ (with some type $\alpha(x) \in \{\sigma_1^d, \dots, \sigma_k^d\}$); *selectors* f_i^j , which extract the j th argument of an f_i -term; and *testers* is_{f_i} , which determine whether a term is an f_i -term. The syntactic categories of terms t and formulas ϕ are defined by the following rules:

$t ::= x$	Variables
$\quad f_i(\bar{t})$	Constructors
$\quad f_i^j(t)$	Selectors
$\phi ::= is_{f_i}(t)$	Testers
$\quad t \approx t$	Equality
$\quad \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi \mid \dots$	Boolean operators

Well-typed terms and formulas are defined as expected, assuming that selectors have type $f_i^j : \sigma_0 \rightarrow \sigma_j$ whenever

$f_i : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma_0$, and testers is_{f_i} expect an argument of type σ_j^d if $f_i : \sigma_j^d$. In the whole paper, we assume that considered expressions are well-typed.

Example 1 (Lists) We show examples in the concrete syntax used in our implementation.

```
1 \sorts {
2   Colour { red; green; blue; };
3   CList { nil; cons(Colour head, CList tail); };
4 }
```

Given variables x of sort CList and y of sort Colour, a formula over this data type is:²

```
1 x.is_cons & y != blue &
2 (x.head = red | x = cons(y, nil))
```

corresponding to the abstract syntax formula

$$is_{cons}(x) \wedge \neg(y \approx blue) \wedge \\ (\text{cons}^1(x) \approx red \vee x \approx \text{cons}(y, nil))$$

Assigning $\{x \mapsto \text{cons}(red, nil), y \mapsto green\}$ satisfies the formula. ■

A constructor term is a ground term t that only consists of constructors (i.e., does not contain selectors or variables). We denote the set of all constructor terms (for some fixed ADT signature) by \mathbb{T} , and the set of all constructor terms of type σ_j^d by $\mathbb{T}_{\sigma_j^d}$. An ADT is well-defined if $\mathbb{T}_{\sigma_j^d}$ is non-empty for all sorts $\sigma_1^d, \dots, \sigma_k^d$, and we will henceforth only consider well-defined ADTs.


Semantics is defined in terms of structures (\mathbb{T}, I) over the universe \mathbb{T} of constructor terms, i.e., constructors are absolutely free. Selectors $f^j : \sigma_0 \rightarrow \sigma_j$ in particular are mapped to total set-theoretic functions $I(f^j) : \mathbb{T}_{\sigma_0} \rightarrow \mathbb{T}_{\sigma_j}$ satisfying $I(f^j)(f(t_1, \dots, t_n)) = t_j$.

III. A VERIFICATION EXAMPLE

As a high-level example, we outline how a simple program operating on the ADT from Example 1 can be verified using our procedures. We represent the program in the form of (constrained) Horn clauses, following the approach taken in several recent verification systems [8], [15]. The result resembles a classical logic program implementing the concatenation of two lists; $C(x, y, r)$ expresses that r is the result of concatenating lists x, y :

```
1 \forall CList y; // (C1)
2 C(nil, y, y)
3 \forall CList x, y, r; \forall Colour c; // (C2)
4 (C(x, y, r) => C(cons(c, x), y, cons(c, r)))
```

As a first property of the program, we can observe that the head of a non-empty result list r has to be the head of one of the arguments x, y :

²In all examples, the link  will take you to the web interface of our SMT solver PRINCESS and directly load the given constraint.

```

1 \forall CList x, y, r; ( // (P1)
2   r != nil & C(x, y, r) ->
3   (r.head = x.head | r.head = y.head))

```

To verify this property, it is enough to find a *model* of the (constrained) Horn clauses (C1), (C2), (P1), i.e., an interpretation of the predicate c that satisfies all three formulas. The predicate c can then be considered as a post-condition (or inductive invariant) that is sufficient to show property (P1). One solution of (C1), (C2), (P1) is to interpret $C(x, y, r)$ as

```

1 C(CList x, CList y, CList r) {
2   r = y | r.head = x.head
3 }

```

which can indeed be observed to satisfy all three clauses. The decision procedure for ADTs defined in the next section can easily check correctness of this model mechanically, after inlining the definition of c , and skolemising away quantifiers.

To find models of clauses like (C1), (C2), (P1) automatically, the principle of *Craig interpolation* can be applied to derivation trees of the clauses, an approach that has been implemented in several model checkers [8], [15]. To support ADTs, which are currently beyond the scope of most model checkers, in Section V we explain how our decision procedure can be extended to Craig interpolation.

Consider now additional clauses computing the list length:

```

1 L(nil, 0) // (C3)
2 \forall CList x; // (C4)
3   \forall Colour c; \forall int n;
4   (L(x, n) -> L(cons(c, x), n+1))

```

We can combine the two programs to state a second property relating concatenation and list length. Concatenating two lists yields a list whose length is the sum of the individual list lengths:

```

1 \forall CList x, y, r; // (P2)
2   \forall int nx, ny, nr; (
3     C(x, y, r) & L(x, nx) & L(y, ny) & L(r, nr)
4     -> nr = nx + ny)

```

To verify this property, as before by showing the existence of a model of (C1), (C2), (C3), (C4), (P2), we need a slightly extended logic providing also an operator for the *size* of ADT terms (Section VI). ADT constraints without size operator are not sufficiently expressive to formulate any model. The size of a term $t \in \mathbb{T}$ is the number of constructor occurrences in t . A model of (C1), (C2), (C3), (C4), (P2), interpreting both the predicate c and L , is then

```

1 C(CList x, CList y, CList r) {
2   \size(x) + \size(y) = \size(r) + 1
3 };
4 L(CList x, int n) {
5   \size(x) = 2*n + 1
6 };

```

Note that the `\size` operator also counts the `nil` symbol, as well as the colour constructors `red`, `green`, `blue`, leading to the stated relationship between the size and the length of a list. The correctness of the model can be checked using the procedure we define in Section VI.

IV. CHECKING ADT SATISFIABILITY BY REDUCTION

We now define our reduction from ADTs to EUF+LIA. Suppose ϕ is an ADT formula as defined in Section II. For sake of presentation, we assume that ϕ has been brought into a *flat* form upfront. A formula ϕ is flat if function symbols (in our case, constructors and selectors) only occur in equations of the form $g(x_1, \dots, x_n) \approx x_0$ (where x_0, \dots, x_n are variables, though not necessarily pairwise distinct), and only in positive positions. Flatness can be established at the cost of introducing a linear number of additional variables.

Example 2 The formula in Example 1 can be flattened by introducing variables $t1, t2 : \text{Colour}$, and $t3 : \text{CList}$:

```

1 x.is_cons & blue = t1 & y != t1 &
2 ((red = t2 & x.head = t2) |
3 (nil = t3 & cons(y, t3) = x))

```

Notation: We need some further notation before we can formally define the reduction. As before, we assume that k sorts $\sigma_1^d, \dots, \sigma_k^d$ and m constructors f_1, \dots, f_m have been fixed. For each sort $\sigma \in \{\sigma_1^d, \dots, \sigma_k^d\}$, we define $\#Ctor_\sigma$ to be the number of constructors of σ :

$$\#Ctor_\sigma = |\{j \mid j \in \{1, \dots, m\} \text{ and } f_j : \sigma\}|$$

Similarly, each constructor f_i with $f_i : \sigma$ is given a unique index $Id_{f_i} \in \{1, \dots, \#Ctor_\sigma\}$ as identifier within its sort σ :

$$Id_{f_i} = |\{j \mid j \in \{1, \dots, i\} \text{ and } f_j : \sigma\}|$$

For each sort $\sigma \in \{\sigma_1^d, \dots, \sigma_k^d\}$, we furthermore need to know the cardinality $|\mathbb{T}_\sigma|$ of the term domain \mathbb{T}_σ . The cardinality can be derived by computing the strongly connected components of the dependency graph induced by the constructors (the graph with sorts $\sigma_1^d, \dots, \sigma_k^d$ as nodes, and edges $\sigma_i^d \rightarrow \sigma_j^d$ whenever there is a constructor with a σ_j^d -sorted argument and result sort σ_i^d). We write $|\mathbb{T}_\sigma| = \infty$ for infinite domains.

A. Definition of the Reduction

Suppose ϕ is a flat formula in negation normal form (NNF) over an ADT as defined in Section II. To translate ϕ to an EUF+LIA formula $\tilde{\phi}$, we introduce a new set of function symbols ranging over integers: for each constructor $f : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma_0$ a new function $\tilde{f} : \mathbb{Z}^n \rightarrow \mathbb{Z}$ with the same arity n ; for each selector $f^j : \sigma_0 \rightarrow \sigma_j$ a unary function $\tilde{f}^j : \mathbb{Z} \rightarrow \mathbb{Z}$; for each sort $\sigma \in \{\sigma_1^d, \dots, \sigma_k^d\}$ a function symbol $ctorId_\sigma : \mathbb{Z} \rightarrow \mathbb{Z}$ to encode testers, and a function $depth_\sigma : \mathbb{Z} \rightarrow \mathbb{Z}$ to ensure acyclicity of terms.

Reduction rules for constructors $f : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma_0$, selectors $f^j : \sigma_0 \rightarrow \sigma_j$, testers is_f , and equations between variables:

$$f(x_1, \dots, x_n) \approx x_0 \implies CtorSpec_f(\tilde{x}_0, \dots, \tilde{x}_n) \quad (1)$$

$$f^j(x) \approx y \implies \tilde{f}^j(\tilde{x}) \approx \tilde{y} \wedge \bigvee_{\substack{g \in \{f_1, \dots, f_m\} \\ g: \sigma_0}} ExCtorSpec_g(\tilde{x}) \quad (2)$$

$$is_f(x) \implies ExCtorSpec_f(\tilde{x}) \quad (3)$$

$$\neg is_f(x) \implies \bigvee_{\substack{g \in \{f_1, \dots, f_m\} \\ g: \sigma_0 \text{ and } g \neq f}} ExCtorSpec_g(\tilde{x}) \quad (4)$$

$$x \approx y \implies \tilde{x} \approx \tilde{y} \quad (5)$$

$$\neg(x \approx y) \implies \neg(\tilde{x} \approx \tilde{y}) \quad (6)$$

The following abbreviations are used, for each constructor $f : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma_0$ and each sort $\sigma \in \{\sigma_1^d, \dots, \sigma_k^d\}$:

$$\begin{aligned} CtorSpec_f(x_0, \dots, x_n) &= \left(\tilde{f}(x_1, \dots, x_n) \approx x_0 \wedge ctorId_{\sigma_0}(x_0) \approx Id_f \wedge \bigwedge_{j=1}^n (\tilde{f}^j(x_0) \approx x_j \wedge depth_{\sigma_0}(x_0) > depth_{\sigma_j}(x_j)) \right) \\ ExCtorSpec_f(x) &= \exists x_1, \dots, x_n. \left(\bigwedge_{j=1}^n In_{\sigma_j}(x_j) \wedge CtorSpec_f(x, x_1, \dots, x_n) \right) \\ In_{\sigma}(x) &= \begin{cases} 0 \leq x < |\mathbb{T}_{\sigma}| & \text{if } |\mathbb{T}_{\sigma}| < \infty \\ true & \text{otherwise} \end{cases} \end{aligned}$$

Table I
RULES FOR REDUCTION OF ADTs TO EUF+LIA

Further, for each variable $x : \sigma$ occurring in ϕ , we introduce an integer-valued variant $\tilde{x} : \mathbb{Z}$.

The actual reduction is defined through the rewriting rules in the upper half of Table I. Since the reduction works differently for positive and negative occurrences of $is_f(x)$ literals, we assume that rules are only applied in positive positions, and handle negation explicitly in the rules (and assume that ϕ is in negation normal form). Rule (1) augments every occurrence of a constructor symbol with corresponding statements about selectors (ensuring that both are inverses of each other); about the index Id_f of the constructor (ensuring that different constructors of the same sort produce distinct values); and about the depth of the constructed term (ensuring that no term can occur as sub-term of itself). Essentially the same translation is done for testers by rule (3), introducing fresh constructor arguments through an existential quantifier. Rule (2) augments each occurrence of a selector with a disjunction stating that the considered term was actually created using one of the constructors of the sort; this is necessary in general since selectors f^j can be applied to terms constructed using constructors other than f (an optimisation is discussed in Section IV-C). Rule (4) asserts that the constructor of a term is different from f , and (5), (6) translate equations by simply renaming variables.

Suppose ϕ^* is the result of exhaustively applying the rules at positive positions in ϕ , and $x_1 : \sigma_1, \dots, x_l : \sigma_l$ are all variables occurring in ϕ , then the reduct of ϕ is defined as

$$\tilde{\phi} = \phi^* \wedge \bigwedge_{i=1}^l In_{\sigma_i}(\tilde{x}_i).$$

Example 3 In the encoded version of the formula from Example 2, all variables and functions range over integers; for readability, we keep the names of all variables. New variables $s1, \dots, s4$ are introduced to eliminate the quantifiers of $ExCtorSpec_f$ expressions through Skolemisation:

```

1 // encoding of x.is_cons
2 cons(s1, s2) = x & ctorId_CList(x) = 1 &
3 head(x) = s1 & tail(x) = s2 & 0 <= s1 & s1 < 3 &
4 depth_CList(x) > depth_Colour(s1) &
5 depth_CList(x) > depth_CList(s2) &
6 // encoding of blue = t1
7 blue = t1 & ctorId_Colour(t1) = 2 &
8 // encoding of y != t1 (unchanged)
9 y != t1 &
10 // encoding of red = t2
11 ((red = t2 & ctorId_Colour(t2) = 0 &
12 // encoding of x.head = t2
13 head(x) = t2 & (
14 // case x.is_nil
15 (nil = x & ctorId_CList(x) = 0) |
16 // case x.is_cons
17 (cons(s3, s4) = x & ctorId_CList(x) = 1 &
18 head(x) = s3 & tail(x) = s4 &
19 0 <= s3 & s3 < 3 &
20 depth_CList(x) > depth_Colour(s3) &
21 depth_CList(x) > depth_CList(s4)))) |
22 // encoding of nil = t3
23 (nil = t3 & ctorId_CList(t3) = 0 &
24 // encoding of cons(y, t3) = x
25 cons(y, t3) = x & ctorId_CList(x) = 1 &
26 head(x) = y & tail(x) = t3 &
27 depth_CList(x) > depth_Colour(y) &

```



```

28 depth_CList(x) > depth_CList(t3))) &
29 // range constraints for x, y, t1, t2, t3
30 // (some of which are just "true")
31 0<=y & y<3 & 0<=t1 & t1<3 & 0<=t2 & t2<3

```

It should be noted that it is not necessary to assume positiveness of the $depth_\sigma$ functions, since the functions are only used to ensure acyclicity of terms by comparing the depth of a term with the depths of its direct sub-terms. In general, although the formula makes use of integer arithmetic, only very simple arithmetic constraints are needed. Up to slight syntactic modifications, all constraints fall into the Unit-Two-Variable-Per-Inequality fragment UTVPI [6], [9], i.e., only inequalities with up to two variables and unit coefficients are needed. The constraints can therefore be both solved and interpolated efficiently (of course, presence of Boolean structure or negation still implies NP-hardness).

B. Correctness of Reduction

Theorem 1 The reduct $\tilde{\phi}$ of a flat ADT formula ϕ in NNF is satisfiable (over EUF+LIA) if and only if ϕ is satisfiable (over an ADT).

Proof: Since reduction preserves the Boolean structure of a formula, and the reduction rules are agnostic of the position at which they are applied, it is enough to prove the theorem for flat conjunctions of literals (i.e., formulas in negation normal form that do not contain disjunctions).

“ \Leftarrow ” (easy direction) Suppose ϕ is satisfiable, with structure (\mathbb{T}, I) and variable assignment β . We construct a family $(\alpha_{\sigma_i^d})_{i=1}^k$ of injective functions as embedding of the domains $\mathbb{T}_{\sigma_i^d}$ into \mathbb{Z} . For i such that $\mathbb{T}_{\sigma_i^d}$ is infinite, $\alpha_{\sigma_i^d}$ can be any bijection $\mathbb{T}_{\sigma_i^d} \rightarrow \mathbb{Z}$; if $\mathbb{T}_{\sigma_i^d}$ is finite, we choose $\alpha_{\sigma_i^d}$ to be a bijection $\mathbb{T}_{\sigma_i^d} \rightarrow \{0, \dots, |\mathbb{T}_{\sigma_i^d}| - 1\}$. Let $\alpha = \bigcup_{i=1}^k \alpha_{\sigma_i^d}$. To satisfy $\tilde{\phi}$, choose variable assignment $\tilde{\beta} = \alpha \circ \beta$, and the interpretation \tilde{I} of constructors and selectors over \mathbb{Z} that is induced by α . Define $\tilde{I}(depth_\sigma)(n)$ to be the depth of the constructor term $\alpha_\sigma^{-1}(n)$, and $\tilde{I}(ctorId_\sigma)(n)$ as the index Id_f of the head symbol f of $\alpha_\sigma^{-1}(n)$ (and arbitrary if $\alpha_\sigma^{-1}(n)$ is undefined).

“ \Rightarrow ” Suppose $\tilde{\phi}$ is satisfiable, with structure (\mathbb{Z}, \tilde{I}) and assignment $\tilde{\beta}$. We construct a set P of relevant integer indices a and corresponding sorts σ in the model, and a mapping $\gamma : P \rightarrow \mathbb{T}$ (with $\gamma(a, \sigma) \in \mathbb{T}_\sigma$ for each $(a, \sigma) \in P$) that can be used to define a variable assignment $\beta(x : \sigma) = \gamma(\tilde{\beta}(\tilde{x}), \sigma)$ to satisfy ϕ . The main difficulty is to ensure that γ is injective, since otherwise disequalities in ϕ might be violated.

We set $P = D \cup D_t$, where D is the set of pairs $(\tilde{\beta}(\tilde{x}), \sigma)$ for variables $x : \sigma$ for which ϕ contains a constructor literal $f(\dots) \approx x$, a selector literal $f^j(x) \approx \dots$, or a (possibly negated) tester $is_f(x)$. The encoding ensures that head symbols and children of terms represented by elements of D are defined by the $ctorId_\sigma$ functions and the selectors; for $(a, \sigma) \in D$, define therefore $dep(a, \sigma) =$

$\langle f, (c_1, \sigma_1), \dots, (c_n, \sigma_n) \rangle$ if $\tilde{I}(ctorId_\sigma)(a) = Id_f$, with $f : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma$, and $c_j = \tilde{I}(f^j)(a)$ for $j \in \{1, \dots, n\}$.

Let D_t contain all pairs (c_i, σ_i) in tuples $dep(a, \sigma) = \langle f, (c_1, \sigma_1), \dots, (c_n, \sigma_n) \rangle$, for any $(a, \sigma) \in D$; as well as pairs $(\tilde{\beta}(\tilde{x}), \sigma) \notin D$ for any further variable $x : \sigma$ in ϕ .

We inductively define a sequence $\gamma_0, \gamma_1, \dots, \gamma_{|P|}$ of partial functions $P \rightarrow \mathbb{T}$:

- 1) let $\gamma_0 = \emptyset$;
- 2) for $i > 0$, if there is $(a, \sigma) \in D$ such that $dep(a, \sigma) = \langle f, (c_1, \sigma_1), \dots, (c_n, \sigma_n) \rangle$, the function γ_{i-1} is defined for each pair (c_j, σ_j) (for $j \in \{1, \dots, n\}$), but γ_{i-1} is not defined for (a, σ) , then let $\gamma_i = \gamma_{i-1} \cup \{(a, \sigma) \mapsto f(\gamma_{i-1}(c_1, \sigma_1), \dots, \gamma_{i-1}(c_n, \sigma_n))\}$.
- 3) for $i > 0$, if case 2) does not apply, pick any pair $(a, \sigma) \in P \setminus D$ for which γ_{i-1} is not defined, and any constructor term $s \in \mathbb{T}_\sigma$ that does not occur in the range of γ_{i-1} yet; choose (a, σ) and s such that the *depth* of s becomes minimal. Let $\gamma_i = \gamma_{i-1} \cup \{(a, \sigma) \mapsto s\}$.

Importantly, the final function $\gamma = \gamma_{|P|}$ is defined for all elements of P , and no two elements of P are mapped to the same term. To see that γ is defined for all elements of P , observe that the use of $depth_\sigma$ functions in the encoding ensures that the dep function is acyclic, i.e., no term can ever be required to contain itself as a sub-term. To see that γ is injective, observe that by definition the choice of s in 3) cannot violate injectivity in γ_i . Different iterations of 2) cannot construct a term $f(\gamma_{i-1}(c_1, \sigma_1), \dots, \gamma_{i-1}(c_n, \sigma_n))$ twice, due to the presence of constructor literals $f(\dots) \approx \tilde{x}$ in $\tilde{\phi}$ that are consistently interpreted. Finally, the fact that case 2) is always preferred over 3) implies that the term $f(\gamma_{i-1}(c_1, \sigma_1), \dots, \gamma_{i-1}(c_n, \sigma_n))$ has to contain the most recently chosen term s from case 3) (if there is any) as a sub-term; this implies that $f(\gamma_{i-1}(c_1, \sigma_1), \dots, \gamma_{i-1}(c_n, \sigma_n))$ is deeper than all terms s previously chosen in case 3), and therefore different from all of them.

It is then possible to choose the variable assignment $\beta(x) = \gamma(\tilde{\beta}(\tilde{x}), \sigma)$ for each variable $x : \sigma$ in ϕ . ■

C. Two Optimisations

The reduction, as presented so far, can be improved in a number of ways. A first optimisation concerns the way selectors are translated to EUF+LIA, rule (2). It can be observed that the disjunction of $ExCtorSpec_g$ literals introduced by rule (2) is in most cases unnecessary, and usually the rule can be simplified to

$$f^j(x) \approx y \implies \tilde{f}^j(\tilde{x}) \approx \tilde{y} \quad (2')$$

This simplification is possible whenever rule (2) is applied to *guarded* selector literals, i.e., whenever $f^j(x) \approx y$ occurs in conjunction with a (positive or negative) test $is_g(x)$, or in conjunction with a constructor literal $g(x_1, \dots, x_n) \approx x$ (in both cases, regardless of whether $f = g$).

Example 4 The effect of this redundancy can be seen in Example 3: given lines 1–5, the disjunction in 14–21 can be simplified to $s3 = s1 \ \& \ s4 = s2$, and can be removed entirely since $s3$ and $s4$ do not occur elsewhere in the formula. ■

Example 5 The full rule (2) is necessary for the following formula over the ADT in Example 1:

```
1  x = cons(x.head, x.tail) <=> x.is_cons  ⌘
```

This is because `x.head` and `x.tail` occur ungarded. ■

As a second optimisation, the treatment of sorts with finite domain can be improved, in particular for sorts that are *enumerations* (i.e., sorts with only nullary constructors). The full EUF encoding is overkill for enumerations, since instead we can map each constructor f directly to the index Id_f :

$$f \approx x_0 \implies Id_f \approx \tilde{x}_0 \quad (1')$$

Similarly, testers in enumerations reduce to simple arithmetic comparisons.

D. Size Increase Caused by the Reduction

The reduction rules replace every literal in a formula ϕ with an expression that is linear in the size n of the considered ADT, so that $|\tilde{\phi}| \in O(n \cdot |\phi|)$. If the ADT is considered as fixed, the reduction is linear.

As an experimental evaluation of the size increase, we applied the procedure (including the optimisations from the previous section) to the 8000 randomly generated ADT benchmarks from [2] (4422 of the benchmarks are unsat). The benchmarks themselves are not very challenging, with the most complicated one solved in around 1 s, and the average solving time of 43 ms dominated by parsing, pre-processing, etc. The average problem sizes, counted as the number of sub-expressions of each formula, were:

After parsing	After reduction	After red. & simpl.
76	337	34

This means that reduction led to an increase in size by a factor of 4.5, but this increase was more than offset by subsequent simplification (using the standard EUF+LIA simplifier implemented in PRINCESS). Analysing further, it turned out that reduction followed by simplification was extremely effective on the unsatisfiable benchmarks: of the 4422 unsatisfiable problems, 4334 could directly be simplified to *false*. The average size of the remaining 3666 problems, after reduction and simplification, was 74, incidentally the same as the average initial size of all benchmarks.

An experimental comparison of our solver with other SMT solvers, on a larger set of benchmarks, is ongoing.

V. CRAIG INTERPOLATION IN (EXTENSIONS OF) ADTs

Since quantifier-free Craig interpolation in EUF+LIA is well understood (e.g., [4]–[6]), the reduction approach can also be leveraged for interpolation. Given an unsatisfiable

conjunction $\phi_A \wedge \phi_B$, the problem of (reverse) interpolation is to find a formula I such that $\phi_A \Rightarrow I$, $\phi_B \Rightarrow \neg I$, and all variables in I are common to ϕ_A and ϕ_B . If ϕ_A, ϕ_B are ADT formulas, it is natural to approach interpolation by first computing an EUF+LIA interpolant \tilde{I} for the reduced conjunction $\tilde{\phi}_A \wedge \tilde{\phi}_B$.

Example 6 An interpolation problem over the list ADT from Example 1 is:

```
1  \part[left]  (x.is_cons & x.tail = z &  ⌘
2              z.is_cons & x.head != z.head)
3  & \part[right] (x = cons(c, cons(c, y)))
```

The only common variable of the two formulas is x , and a solution of the interpolation problem is the disequality $x.head != x.tail.head$. Note that this formula is a correct interpolant even though the selectors are ungarded. ■

To translate \tilde{I} back to an ADT interpolant I , three main points have to be addressed. First, since all ADT sorts are translated to integers, the formula \tilde{I} might contain arithmetic operations on ADT terms that cannot easily be mapped back to the ADT world. This turns out to be a non-issue for infinite ADT sorts, since reduction does not make use of arithmetic operations for terms over infinite sorts (indeed, equivalently every infinite ADT sort σ^d could be mapped to a fresh uninterpreted sort $\tilde{\sigma}^d$). The situation is different for finite sorts, where predicates In_σ from Table I represent cardinality constraints that can contribute to unsatisfiability of a formula. One solution is the optimisation discussed in Section IV-C: by defining a *fixed* mapping of terms in finite domains to integers, translation of interpolants back to ADT formulas is significantly simplified.³

Second, the functions $ctorId_\sigma$ introduced by the reduction are not valid ADT operations, and have to be translated back to testers (which can be done quite easily).

Third, interpolants might also mention $depth_\sigma$ operations, which have no direct correspondence in the original ADTs theory. Instead of devising ways how to eliminate such operations, we decide to embrace them instead as a useful extension of ADTs, and adapt our reduction method accordingly. Since depth is but one measure that can be used to ensure acyclicity, the next sections therefore discuss how we can reason about ADTs with *size constraints*.

VI. SOLVING ADTs WITH SIZE CONSTRAINTS

We now consider ADT formulas extended with constraints about term size. The *size* $|t|$ of a term $t \in \mathbb{T}$ is the number of constructor occurrences in t . The resulting formal language is an extension of the language defined in Section II:

$$\phi ::= \dots \mid \phi_{\text{Pres}}(|t_1|, \dots, |t_n|) \quad \text{Size constraints}$$

³In our implementation, such fixed mapping is currently only done for enumerations, not for other finite ADT sorts.

where $\phi_{\text{Pres}}(|t_1|, \dots, |t_n|)$ is any Presburger formula about the size of ADT terms t_1, \dots, t_n .

Example 7 Consider the ADT in Example 1, and a variable x of sort `CLIST`. The formula

```
1 \size(x) = 3 & x.head = blue
```

has the satisfying assignment $x \mapsto \text{cons}(\text{blue}, \text{nil})$, and this assignment is unique. In contrast, the formula

```
1 \size(x) % 2 = 0
```

is unsatisfiable, since the size of any list term is odd (term size does not exactly coincide with the length of a list). ■

To extend our reduction approach to formulas with size constraints, there are two main issues that have to be addressed: (i) constructor terms might not exist for all sizes $n \in \mathbb{N}_{\geq 1}$, and (ii) even if terms of some size $n \in \mathbb{N}_{\geq 1}$ exist, there might be too few of them to satisfy a formula.

Example 8 Consider the ADT of positive natural numbers:

```
1 \sorts {
2   Nat { one; succ(Nat pred); };
3 }
```

For every size $b \in \mathbb{N}_{\geq 1}$ there is exactly one constructor term t with $|t| = b$. This implies unsat. of the formula

```
1 \size(x) = 3 & \size(y) = 3 & x != y
```

A. Reduction and Incremental Unfolding

We address issue (i) noted above by reasoning globally about possible term sizes within an ADT. For an ADT sort σ_j^d , we define $\mathbb{S}_{\sigma_j^d} = \{|t| \mid t \in \mathbb{T}_{\sigma_j^d}\} \subseteq \mathbb{N}$ to be the *size image* of the term set $\mathbb{T}_{\sigma_j^d}$, i.e., the set of term sizes in $\mathbb{T}_{\sigma_j^d}$. The size image turns out to be a special case of the *Parikh image* of a context-free language, since an ADT can be interpreted as a context-free grammar over a singleton alphabet (by considering every sort as a non-terminal symbol, and mapping every constructor to the unique letter in the singleton alphabet). This implies that $\mathbb{S}_{\sigma_j^d}$ is semi-linear, and that a representation of the set in the form of an existential Presburger formula can be derived from the set of constructors in linear time [18].

Table II shows how the reduction from Section IV-A (and Table I) is augmented to deal with size constraints. Instead of the depth_σ functions, for each sort $\sigma \in \{\sigma_1^d, \dots, \sigma_k^d\}$ a function $\text{size}_\sigma : \mathbb{Z} \rightarrow \mathbb{Z}$ representing term size is introduced, and the CtorSpec_f constraints are changed accordingly; and an additional reduction rule (7) is introduced to handle equations $|x| \approx y$ with the size operation. Rule (7) adds constraints $y \in \mathbb{S}_\sigma$ to ensure that only genuine term sizes are considered, assuming implicitly that the size image \mathbb{S}_σ is represented as a Presburger formula.

The resulting modified reduction approach is sound for checking unsatisfiability of ADT formulas:

Lemma 1 If the reduct $\tilde{\phi}$ of a flat ADT formula ϕ in NNF with size constraints is unsatisfiable, then ϕ is unsatisfiable.

Reduction does not directly give rise to a decision procedure for ADT constraints with size constraints, in contrast to the situation without size. This is because reduction does not precisely translate the *number* of terms for each size $n \in \mathbb{N}_{\geq 1}$ (issue (ii) from above). We can observe that the reduct $\tilde{\phi}$ of the formula ϕ in Example 8 is satisfiable, while ϕ is unsatisfiable, showing that reduction alone is *not* sound for satisfiability (unsurprisingly).

Different approaches exist to establish soundness also for satisfiability, in particular the extraction of *length constraint completion* formulas [19] that precisely define term sizes with sufficiently many distinct terms. We follow the approach of incrementally unfolding (aka. unrolling) from [17], which is quite flexible, and complete in many relevant cases.

Let ϕ again be a (flat and NNF) ADT formula with size constraints. We construct unfolding sequences ϕ_0, ϕ_1, \dots by setting $\phi_0 = \phi$, and for each $i \geq 1$ deriving ϕ_i by unfolding one ADT variable $x : \sigma$ that occurs in ϕ_{i-1} . If f_1, \dots, f_n are all constructors of the considered ADT, we set

$$\phi_i = \phi_{i-1} \wedge \bigvee_{\substack{j \in \{1, \dots, n\} \\ f_j : \sigma}} f_j(x_1^j, x_2^j, \dots) \approx x$$

with fresh sorted argument variables $x_1^1, x_2^1, \dots, x_1^2, x_2^2, \dots$.

In practice, unfolding will usually happen incrementally: the next variable to unfold is selected based on a model of the previous partial unfolding ϕ_{i-1} , until enough terms have been constructed to obtain a genuine model of ϕ , or unsatisfiability is detected.

Lemma 2 Let ϕ_0, ϕ_1, \dots be an unfolding sequence for ϕ , and for each $i \in \mathbb{N}$ let U_i be the set of variables unfolded in ϕ_i (i.e., $U_0 = \emptyset$, and $U_i = U_{i-1} \cup \{x\}$ if ϕ_i was derived by unfolding x in ϕ_{i-1}). Then for any $i \in \mathbb{N}$:

- 1) if $\tilde{\phi}_i$ is unsatisfiable (over EUF+LIA) then ϕ is unsatisfiable (over ADTs with size);
- 2) if $\tilde{\phi}_i$ is satisfied by a model \tilde{M} and assignment $\tilde{\beta}$, such that for every ADT variable $x : \sigma$ in ϕ_i there is a variable $y \in U_i$ with $y : \sigma$ and $\text{val}_{\tilde{M}, \tilde{\beta}}(\tilde{x}) = \text{val}_{\tilde{M}, \tilde{\beta}}(\tilde{y})$, then ϕ is satisfiable (over ADTs with size).

Proof: 1) follows directly from Lemma 1.

2) Models over EUF+LIA can be translated to ADT models like in the proof of Theorem 1 “ \implies ”. It can be noted that case 3) in the proof never applies due to the assumption that all variables are mapped to unfolded terms. ■

Example 9 In Example 8, unsatisfiability is detected after unfolding x and y three times each. ■

Additional reduction rule for size expressions, assuming x is a variable of sort $\sigma \in \{\sigma_1^d, \dots, \sigma_k^d\}$, and y a variable of sort \mathbb{Z} :

$$|x| \approx y \implies \text{size}_\sigma(\tilde{x}) \approx y \wedge y \in \mathbb{S}_\sigma \quad (7)$$

Compared to Table I, for each constructor $f : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma_0$ the abbreviation CtorSpec_f is replaced with $\text{CtorSpec}'_f$:

$$\text{CtorSpec}'_f(x_0, \dots, x_n) = \left(\tilde{f}(x_1, \dots, x_n) \approx x_0 \wedge \text{ctorId}_{\sigma_0}(x_0) \approx \text{Id}_f \wedge \bigwedge_{j=1}^n (\tilde{f}^j(x_0) \approx x_j \wedge \text{size}_{\sigma_j}(x_j) \in \mathbb{S}_{\sigma_j}) \wedge \text{size}_{\sigma_0}(x_0) \approx 1 + \sum_{j=1}^n \text{size}_{\sigma_j}(x_j) \right)$$

Table II
ADDITIONAL RULES FOR REDUCTION OF ADTs WITH SIZE CONSTRAINTS TO EUF+LIA

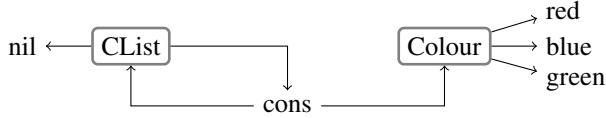


Figure 1. Dependency graph for the list ADT from Example 1

As the next example shows, however, unfolding is not always enough to show unsatisfiability of a formula. The next sections will therefore formulate a sufficient and necessary criterion for termination of unfolding.

Example 10 With the ADT from Example 8, the formula

1 `\size(x) = \size(y) & x != y` ✎

is unsatisfiable, but cannot be shown to be unsatisfiable with a finite number of unfolding steps. ■

B. Completeness and Incompleteness of Unfolding

We give a precise characterisation of the ADTs for which unfolding will allow us to eventually detect (un)satisfiability of a formula, and therefore gives rise to a decision procedure. As identified in [17], the essential property of an ADT (resp., of the considered catamorphism, which in our case is the size function) is *sufficient surjectivity*, implying that ADTs are sufficiently populated to satisfy disequalities in a formula: the number of terms of size b grows unboundedly when b tends to infinity. We write \mathbb{T}_σ^k for the set of constructor terms of ADT sort σ and size k , i.e., $\mathbb{T}_\sigma^k = \{t \in \mathbb{T}_\sigma^k \mid |t| = k\}$.

Definition 1 An ADT sort σ^d is *expanding* if for every natural number $n \in \mathbb{N}$ there is a bound $b \in \mathbb{N}$ such that for every $b' \geq b$ either $\mathbb{T}_{\sigma^d}^{b'} = \emptyset$ or $|\mathbb{T}_{\sigma^d}^{b'}| \geq n$. An ADT is expanding if each of its sorts $\sigma_1^d, \dots, \sigma_k^d$ is expanding.

Example 11 An example of an ADT that is *not* expanding are the natural numbers (Example 8): for every size $b \in \mathbb{N}_{\geq 1}$ there is exactly one constructor term t with $|t| = b$. ■

Lemma 3 Systematic unfolding terminates (i.e., in every unfolding sequence ϕ_0, ϕ_1, \dots in which every variable is eventually unfolded, eventually one of the cases of Lemma 2

applies) for all formulas ϕ if and only if the considered ADT is expanding.

Proof sketch: “ \implies ” Example 10 generalises to arbitrary non-expanding ADTs: for every non-expanding sort σ there is a constant $c \in \mathbb{N}$ and an infinite semi-linear set $S \subseteq \mathbb{S}_\sigma$ such that $|\mathbb{T}_\sigma^b| < c$ for all $b \in S$. The existence of c, S follows from the proof of Theorem 2 below.

“ \impliedby ” Consider first the case of ϕ being a conjunction of disequalities and a size constraint $\phi_{\text{Pres}}(|x_1|, \dots, |x_n|)$. Since the ADT is expanding, satisfiability of ϕ reduces to the question whether the size images of the x_i -domains contain elements large enough, and compatible with ϕ_{Pres} , that all disequalities can be satisfied. Systematic unfolding of x_1, \dots, x_n will add size image constraints $|x'| \in \mathbb{S}_\sigma$ for all sub-terms, and either find a set of satisfying term sizes (and corresponding terms), or conclude unsatisfiability because the conjunction of size images and size constraint ϕ_{Pres} becomes inconsistent.

Adding constructor, selector, or test literals does not change the argument, since solutions of such literals can be represented in the form of a most-general unifier [17]. ■

Non-expandingness turns out to be a corner case: all non-expanding ADTs more or less look like the natural numbers (Example 8), and most other practical ADTs are expanding. For instance, both ADT sorts in Example 1 expand.

C. Effective Characterisation of Expanding ADTs

To characterise expanding ADTs, we first make the simplifying assumption that all ADT sorts σ_j^d contain at least two constructor terms; sorts with only a single term can obviously be eliminated easily from a constraint. As a further piece of notation, we need a relativised version of the size image: for an ADT sort σ_j^d and a constructor f , we write

$$\mathbb{S}_{\sigma_j^d}^f = \{|t| \mid t \in \mathbb{T}_{\sigma_j^d}, \text{ and } t \text{ does not start with } f\}$$

for the size image restricted to terms with head symbol $\neq f$.

Consider then the bipartite dependency graph $D = (V, E)$ with vertices $V = \{\sigma_1^d, \dots, \sigma_k^d\} \cup \{f_1, \dots, f_m\}$ being sorts

and constructors, and the edge set

$$E = \left\{ (\sigma_0, f_j), (f_j, \sigma_1), \dots, (f_j, \sigma_n) \mid j \in \{1, \dots, m\}, f_j : \sigma_1 \times \dots \times \sigma_n \rightarrow \sigma_0 \right\}$$

Fig. 1 gives the graph for the list ADT in Example 1.

Theorem 2 An ADT is *not* expanding if and only if the graph D contains a simple cycle $C = \sigma^1 \rightarrow f^1 \rightarrow \sigma^2 \rightarrow f^2 \rightarrow \dots \rightarrow f^n \rightarrow \sigma^1$ with the following properties:

- 1) C is the only path from σ^1 to itself, i.e., every cycle starting and ending in σ^1 is a repetition of C ;
- 2) all constructors f^1, f^2, \dots, f^n on C are unary;
- 3) the cycle C unboundedly contributes to the size image \mathbb{S}_{σ^1} , i.e.,

$$\forall k. \mathbb{S}_{\sigma^1} \neq \{0, \dots, k\} \cdot n + \bigcup_{i=1}^n (\mathbb{S}_{\sigma^i}^{f^i} + i - 1).$$

The characterisation theorem implies that every non-expanding ADT has a set of cyclically connected sorts S_1, \dots, S_n , each of which might contain further constructors c_{1_1}, c_{1_2}, \dots that do not lead back to S_1, \dots, S_n :

```

1 \sorts {
2   // ...
3   S1 { f1(S2 s2); c1_1; c1_2; /* ... */ };
4   S2 { f2(S3 s3); c2_1; c2_2; /* ... */ };
5   // ...
6   Sn { fn(S1 s1); cn_1; cn_2; /* ... */ };
7   // ...
8 }
```

The conditions of the theorem are clearly satisfied for the ADT of natural numbers (Example 8). Condition 3) is satisfied whenever $\mathbb{S}_{\sigma^i}^{f^i}$ is finite for all $i \in \{1, \dots, n\}$, but there are more subtle situations:

Example 12 We extend the list ADT from Example 1 by adding two further sorts:

```

1 \sorts {
2   S1 { f1(S2 s2); };
3   S2 { f2(S1 s1); null; col(CList list); };
4 }
```

The domain of sort S_1 contains terms of any size greater than one. However, while the number of terms of size $2k+1$ grows exponentially with k , there is exactly one term of size $2k$ for every k , proving non-expandingness.

A cycle of length 3, in contrast, yields an expanding ADT:

```

1 \sorts {
2   S1 { f1(S2 s2); };
3   S2 { f2(S3 s3); };
4   S3 { f3(S1 s1); null; col(CList list); };
5 }
```

We can note that condition 3) of Theorem 2 now fails. ■

Before we can prove Theorem 2, we need some further results about ADTs. Consider constructor terms $t[\bullet]$ with a

unique hole \bullet ; such terms can alternatively be seen as terms with a single occurrence of a sorted variable. Composition of terms with holes is defined as $t_1[\bullet] \circ t_2[\bullet] = t_1[t_2[\bullet]]$. Two terms $t_1[\bullet], t_2[\bullet]$ with holes are *incomparable* if $t_1[s_1] \neq t_2[s_2]$ for all constructor terms s_1, s_2 of the right sort. The size $|t[\bullet]|$ of a term with hole is the number of constructor symbol occurrences in t , i.e., the hole does not count. This implies $|t_1[\bullet] \circ t_2[\bullet]| = |t_1[\bullet]| + |t_2[\bullet]|$.

Lemma 4 Suppose an ADT sort σ contains two incomparable constructor terms $t_1[\bullet], t_2[\bullet]$ with holes \bullet of sort σ . Then σ expands.

Proof: Fix some $n \in \mathbb{N}$. We need to show that there is a bound $b \in \mathbb{N}$ ensuring $\mathbb{T}_\sigma^{b'} = \emptyset$ or $|\mathbb{T}_\sigma^{b'}| \geq n$ for every $b' \geq b$. Observe that for every $g \geq n$ there are $> n$ pairwise incomparable terms with holes $t_1^0 \circ t_2^g \circ t_1^g, t_1^1 \circ t_2^g \circ t_1^{g-1}, \dots, t_1^g \circ t_2^g \circ t_1^0$, all of which have size $g \cdot (|t_1| + |t_2|)$. The size image $\mathbb{S}_\sigma \subseteq \mathbb{N}$ can be represented as a finite union of arithmetic progressions, $\mathbb{S}_\sigma = \bigcup_{i=1}^l \{a_i + k \cdot b_i \mid k \in \mathbb{N}\}$ with $a_i, b_i \in \mathbb{N}$ for $i \in \{1, \dots, l\}$. For each $i \in \{1, \dots, l\}$ with $b_i > 0$, assuming that $k \geq n \cdot (|t_1| + |t_2|)$ we can then pick $g = b_i \cdot n$, and some term $t : \sigma$ of size $a_i + (k - n \cdot (|t_1| + |t_2|)) \cdot b_i$, and obtain $> n$ pairwise distinct terms

$$(t_1^0 \circ t_2^g \circ t_1^g)[t], (t_1^1 \circ t_2^g \circ t_1^{g-1})[t], \dots, (t_1^g \circ t_2^g \circ t_1^0)[t]$$

that are all of size $a_i + (k - n \cdot (|t_1| + |t_2|)) \cdot b_i + g \cdot (|t_1| + |t_2|) = a_i + k \cdot b_i$. This implies that there is also a global bound $b \in \mathbb{N}$ such that $\mathbb{T}_\sigma^{b'} = \emptyset$ or $|\mathbb{T}_\sigma^{b'}| \geq n$ for $b' \geq b$. ■

Proof of Theorem 2: “ \Rightarrow ” Suppose an ADT is not expanding, which means that there is a non-expanding sort σ . Choose σ such that whenever $\sigma \xrightarrow{*} \sigma'$ in the D -graph, and σ' is non-expanding as well, then σ' is in the same strongly connected component (SCC) as σ ; this is possible because the SCCs of D form a DAG. Then \mathbb{S}_σ has to be infinite (otherwise σ would be expanding), and there is a simple D -path $C = \sigma^1 \rightarrow f^1 \rightarrow \sigma^2 \rightarrow f^2 \rightarrow \dots \rightarrow f^n \rightarrow \sigma^1$ with $\sigma^1 = \sigma$ (otherwise there would be a non-expanding sort σ' reachable from σ , but not in the same SCC).

We show that C satisfies the conditions of the theorem. 1) holds because if there was a second path C' from σ to itself that is not a repetition of C , then both paths could be translated to incomparable σ -terms $t_1[\bullet], t_2[\bullet]$ with σ -sorted holes \bullet , and by Lemma 4 the sort σ would be expanding. The same argument implies that 2) holds: if any of the constructors f_i had multiple arguments, incomparable terms $t_1[\bullet], t_2[\bullet]$ could be derived (since, by assumption, every sort contains at least two constructor terms).

Suppose finally that 3) does not hold, i.e., for some $k \in \mathbb{N}$

$$\mathbb{S}_\sigma = \{0, \dots, k\} \cdot n + \bigcup_{i=1}^n (\mathbb{S}_{\sigma^i}^{f^i} + i - 1)$$

This would imply that from some sort σ^i a non-expanding sort σ' is reachable, by following a constructor other than f^i .

By choice of σ , then σ' has to be in the same SCC as σ , therefore there is a path from σ' to σ , and condition 1) would be violated.

“ \Leftarrow ” Suppose there is a cycle C satisfying 1)–3). Note that due to 1) and 2) we have the equality

$$\mathbb{S}_\sigma = \mathbb{N} \cdot n + \underbrace{\bigcup_{i=1}^n (\mathbb{S}_{\sigma^i}^{f^i} + i - 1)}_R$$

Together with 3), this means $\mathbb{N} \cdot n + R \neq \{0, \dots, k\} \cdot n + R$ for every $k \in \mathbb{N}$. Because R is a semi-linear set, then there has to be a finite subset $S \subseteq R$ such that for infinitely many points $s \in \mathbb{S}_\sigma$ we have $\{x \in R \mid s \in \mathbb{N} \cdot n + x\} \subseteq S$. Because the set $\{t \in \mathbb{T}_\sigma \mid |t| \in S\}$ of terms is finite,⁴ this immediately implies that the sort σ is not expanding. ■

VII. CONCLUSIONS

At the moment we are exploring applications and further extensions of our approach. We are in the process of integrating our procedure into the model checker ELDARICA [15] to handle implication checks and interpolation for ADTs; this also requires combination with other data types, and in the long run likely interpolation heuristics. It is also frequently necessary to combine ADTs with quantifier reasoning and recursively defined functions, a direction that requires further work. Finally, as a side-effect of Theorem 2, there is a simple way to achieve termination also for non-expanding ADTs, namely by replacing the cycle with an explicit counter ranging over a built-in type of natural numbers.

Acknowledgements: Rümmer was supported by the Swedish Research Council under grant 2014-5484.

REFERENCES

- [1] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
- [2] Clark Barrett, Igor Shikanian, and Cesare Tinelli. An abstract decision procedure for a theory of inductive data types. *JSAT*, 3(1-2):21–46, 2007.
- [3] Clark Barrett and Cesare Tinelli. CVC3. In *CAV*, volume 4590 of *LNCS*, pages 298–302. Springer, July 2007.
- [4] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. Beyond quantifier-free interpolation in extensions of Presburger arithmetic. In *VMCAI*, *LNCS*, pages 88–102. Springer, 2011.
- [5] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In *SPIN*, pages 248–254, 2012.
- [6] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Interpolant generation for UTVPI. In *CADE*, volume 5663 of *LNCS*, pages 167–182. Springer, 2009.
- [7] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
- [8] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012.
- [9] Joxan Jaffar, Michael J. Maher, Peter J. Stuckey, and Roland H. C. Yap. Beyond finite domains. In *PPCP*, volume 874 of *LNCS*, pages 86–94. Springer, 1994.
- [10] Deepak Kapur, Rupak Majumdar, and Calogero G. Zarba. Interpolation for data structures. In *SIGSOFT’06/FSE-14*, pages 105–116, New York, NY, USA, 2006. ACM.
- [11] Zohar Manna, Nikolaj Bjørner, Anca Browne, Edward Y. Chang, Michael Colón, Luca de Alfaro, Harish Devarajan, Arjun Kapur, Jaejin Lee, Henny Sipma, and Tomás E. Uribe. STeP: The Stanford temporal prover. In *TAPSOFT*, volume 915 of *LNCS*, pages 793–794. Springer, 1995.
- [12] Derek C. Oppen. Reasoning about recursively defined data structures. *J. ACM*, 27(3):403–411, July 1980.
- [13] Andrew Reynolds and Jasmin Christian Blanchette. A decision procedure for (co)datatypes in SMT solvers. *J. Autom. Reasoning*, 58(3):341–362, 2017.
- [14] Philipp Rümmer. A constraint sequent calculus for first-order logic with linear integer arithmetic. In *LPAR*, volume 5330 of *LNCS*, pages 274–289. Springer, 2008.
- [15] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for Horn-clause verification. In *CAV*, volume 8044 of *LNCS*, pages 347–363. Springer, 2013.
- [16] Viorica Sofronie-Stokkermans. Locality results for certain extensions of theories with bridging functions. In *CADE*, volume 5663 of *LNCS*, pages 67–83. Springer, 2009.
- [17] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. *SIGPLAN Not.*, 45(1):199–210, January 2010.
- [18] Kumar Neeraj Verma, Helmut Seidl, and Thomas Schwenk. On the complexity of equational Horn clauses. In *CADE*, volume 3632 of *LNCS*, pages 337–352. Springer, 2005.
- [19] Ting Zhang, Henny B. Sipma, and Zohar Manna. Decision procedures for term algebras with integer constraints. *Inf. Comput.*, 204(10):1526–1574, 2006.

⁴This property breaks down when ADTs are combined with other infinite data types, e.g., lists over \mathbb{Z} . In this case condition 3) has to be modified.