Relational Analysis of Algebraic Datatypes

Viktor Kuncak
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, MA 02139
vkuncak@mit.edu

Daniel Jackson
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, MA 02139
dnj@mit.edu

ABSTRACT

We present a technique that enables the use of finite model finding to check the satisfiability of certain formulas whose intended models are infinite. Such formulas arise when using the language of sets and relations to reason about structured values such as algebraic datatypes. The key idea of our technique is to identify a natural syntactic class of formulas in relational logic for which reasoning about infinite structures can be reduced to reasoning about finite structures. As a result, when a formula belongs to this class, we can use existing finite model finding tools to check whether the formula holds in the desired infinite model.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—model checking, formal methods; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—logics of programs, mechanical verification, specification techniques

General Terms

Design, Reliability, Theory, Verification

Keywords

model checking, model finding, algebraic datatypes, constraint solving, transitive closure logic

1. INTRODUCTION

A new kind of analysis has become popular in the last decade in which a system is examined by considering all small cases within some bound. The rationale is that flaws are revealed more readily by this method than by conventional testing: exhausting a huge space of small cases works better than considering a much smaller suite of cases, even if it includes larger ones.

Model checking is the preeminent example of this approach, and bounds the set of reachable states and sometimes also the length of execution traces. The success of model checking in hardware verification has generated great interest in applying it to software. Most model checkers, though, offer only rudimentary support for data structures, so applications of model checking to software until now have focused on control properties, and data has either been ignored or abstracted away.

To handle data structures effectively within this context, a reduction to small cases is needed. With such a reduction, no special abstractions for data would be needed, and the same bounding mechanism used for trace length, for example, could be applied to the size of data structures.

How should data structures be represented in such an analysis? A relational representation is very attractive, because it fits both the analyses that are widely used at the low level, and the object-oriented view of a program at the high level. Symbolic model checkers [20] already represent the state as a bit vector; the adjacency matrix representation of a relation is therefore easily integrated. In the object-oriented view of program state, the heap is a graph, with objects as nodes and fields of objects as edges between objects—in other words, a collection of relations, one per field [18]. This view is common in object-oriented development [39], because it helps postpone the allocation of state (in the form of relations) to classes (in the form of fields), and is attractive in program analysis because it's simple and easily accounts for sharing (a shared object simply being in the range of two relations).

An important question to ask, therefore, is whether this relational viewpoint can accommodate a general theory of data structures. Can arbitrary structural properties be naturally expressed and analyzed by the small case approach? This question is not only of theoretical interest. It has arisen repeatedly amongst advanced users of one tool, the Alloy language and its associated analyzer [1], as they have discovered occasional scenarios in which Alloy's relational encoding does not seem to capture their intuitions about data structures.

This question has implications not only for Alloy, but more generally for any tool that relies on small case analysis of a relational encoding. This includes not only model checkers (such as SMV [20] and NuSMV [6]), but also specification analysis tools based on constraints (such as ProB [26] and the Bremen USE tool [10]), and indeed potentially to any tool that encodes data relationally.

To frame the problem rigorously, a characterization of data structures independent of the relational viewpoint is needed. For this purpose, we use the theory of algebraic datatypes, which corresponds to the way programmers think about structured values, and is the basis for their definition in several programming languages.

We start by explaining the standard encoding of algebraic datatypes using relations in first order logic. This encoding is faithful, but it suffers from a major drawback: it includes an axiom that requires all models of a formula to be infinite. Consequently, an analysis based on finite cases cannot be applied. To remedy this, we remove the offending axiom. Surprisingly, most analyses performed in its absence are still sound, although some analyses will produce spurious counterexamples.

The principal contribution of this paper is a simple syntactic criterion that guarantees that a formula being analyzed will not suffer from spurious counterexamples. The criterion is easy to understand, and could be applied automatically by a tool, warning the user when an analysis on the relational encoding might produce results that do not correspond to the full theory of algebraic datatypes.

2. AN EXAMPLE

In this section, we motivate the problem with an example of a relational encoding of a simple algebraic datatype. We show how the omission of a *generator* axiom can cause spurious counterexamples, but its inclusion results in inconsistency, making the encoding useless. The challenge is to determine under what conditions the axiom can be omitted while remaining faithful to the theory of algebraic datatypes.

The example will be given in Alloy [19], a modelling language based on a simple first-order logic with relational operators. Although our work was motivated by Alloy, our results apply more broadly, and the rest of the paper presents our theory in a transitive closure logic that has no Alloy-specific features.

A datatype for lists would be declared in a language such as ML [34] like this:

```
datatype List = Nil | Cons of Element * List
```

where `List` is the datatype being declared, `Element` is the type corresponding to the elements, and `Nil` and `Cons` are *constructors*, with no arguments and two arguments respectively.

In Alloy, `List` and `Element` might be represented as top-level sets (called *signatures* in Alloy):

```
sig Element {}
sig List {}
```

`Nil` is a singleton set – the set containing the empty list:

```
one sig Nil extends List {}
```

`Cons` is represented by a set `Cons` and two *selectors*, `elt` and `rest`:

```
sig Cons extends List {
    elt: Element,
    rest: List
}
```

The `extends` syntax makes `Cons` a subset of `List`, disjoint from `Nil`. The selectors are semantically just relations from the set `Cons` to the sets `Element` and `List` respectively, implicitly constrained to be total functions by Alloy's declaration syntax.

The function `cons` that appends an element to the front of a list can be written as an Alloy predicate:

```
pred cons(e: Element, l: List, c: Cons) {
   c.elt = e and c.rest = l
}
```

which associates with an element `e` and a list `l` any object `c` in `Cons` such that `e` and `l` are the element and rest components of `c`.

Let's consider checking some putative theorems, called *assertions* in Alloy. This assertion says that the first element of a list resulting from an application of `cons` is the same element that was presented as an argument:

```
assert ElementIsArg {
  all e: Element, l: List, c: Cons |
    cons(e, l, c) => e = c.elt
}
```

This holds trivially—the consequent being contained in the hypothesis. When checked by the Alloy Analyzer, it yields no counterexamples. In contrast, suppose we check the assertion that `cons` is deterministic:

```
assert ConsDeterministic {
  all e: Element, l: List, c, c': Cons |
    (cons (e, l, c) and cons(e, l, c')) => c = c'
}
```

This is invalid, and the Alloy Analyzer will give a counterexample such as:

```
List = {L0, L1, L2}; Cons = {L1, L2}; Nil = {L0};
Element = {E0};
elt = {(L1, E0), (L2, E0)};
rest = {(L1, L0), (L2, L0)};
e = {E0}; l = {L0};
c = {L1}; c' = {L2}
```

in which the two applications of `cons` produce distinct lists `L1` and `L2` that are structurally identical.

This might be acceptable for some applications, but if we wanted to model the kind of list used in languages such as ML, in which equality is structural (and identity of cells therefore cannot be observed), we could add an axiom (called a *fact* in Alloy):

```
fact Canonical {
  all l, l': List |
    l.elt = l'.elt and l.rest = l'.rest => l = l'
}
```

requiring that structurally identical lists have the same identity, ensuring that the assertion `ConsDeterministic` is now valid.

Continuing with our exploration, we might notice that in some cases Alloy generates cyclic lists as counterexamples of properties. To rule these out we might introduce another fact

```
fact Acyclic {
  no l : List | l in l.^rest
}
```

ensuring that `rest` is an acyclic relation.

So far, so good. Consider, however, an assertion claiming that `cons` is total:

```
assert ConsTotal {
  all l: List, e: Element | some c: Cons |
    cons(e, l, c)
}
```

Given our intuition about algebraic datatypes, we expect this assertion to be valid. But in the relational setting, given the facts stated so far, the assertion `ConsTotal` is actually invalid. The Alloy Analyzer will generate a counterexample such as:

```
List = {L0}; Cons = {}; Nil = {L0};
Element = {E0};
elt = {};
rest = {};
e = {E0}; l = {L0}
```

The problem, roughly speaking, is that the Analyzer is free to construct a counterexample in which there aren't enough lists; in this case, only the empty list is available.

Suppose, following our previous strategy, we attempt to add a fact to rule out counterexamples of this form:

```
fact Generator {
  all l: List, e: Element | some c: Cons |
      c.elt = e and c.rest = l
}
```

The `Generator` axiom ensures that the selector relations are complete: for any combination of list and element, it requires the existence of list for which they are components. This will indeed rule out the counterexample above; in this trivial example, the assertion `ConsTotal` is barely different from `Generator` itself. Unfortunately, however, adding this axiom rules out *all* (nonempty) counterexamples, even to a manifestly false assertion. For example, the assertion

```
assert NoDistinctElements {
  all e, e': Element | e = e'
}
```

will be valid, and will have no counterexamples. The problem is that the generator axiom has no finite models, and is therefore *inconsistent* in a setting in which only finite models are considered.

The key idea of this paper is that a finite checker cannot incorporate this axiom, but must nevertheless be able to handle algebraic datatypes. The question then is what class of formulas have the same models whether or not this axiom is included. The contribution of this paper is a characterization of a class of such formulas that is both expressive and easily checked syntactically.

Assertion `ConsTotal`—and, not surprisingly, `Generator`, the generator axiom itself—will not be in this class. The culprit is the innermost quantification. The nesting of quantifiers is not in itself problematic; rather, the problem is that the quantification is not bounded. If, instead, it were bounded by an expression in terms of variables bound in an outer quantifier, no spurious counterexample would be generated, even in the absence of the generator axiom. For example, the assertion

```
assert Deconstruct {
  all c: Cons | some l: c.*rest, e: Element |
    cons(e, l, c)
}
```

saying that each list is the result of an application of cons to some sublist (and in which `c.*rest` is the application of the reflexive transitive closure of `rest` to `c`, giving the set of `c`'s sublists), is valid, as expected.

3. LOGIC AND ALGEBRAIC DATATYPES

This section introduces our two formalisms: term algebras, a theory of algebraic datatypes, and a first-order logic with transitive closure. We show how a term algebra can be straightforwardly translated into first-order logic, using four axioms. One of these is a generator axiom that causes all models to be infinite. In the following section, we establish our main result about when this axiom can be omitted.

Throughout the paper, we use binary trees as the canonical example of algebraic datatypes. Being simple and familiar, trees serve well as a pedagogical example. In addition, because trees can represent all other algebraic datatypes, there is no loss of generality.

Algebraic datatypes and term algebras. We consider structures that contain two kinds of values, or *sorts*: 1) a Tree sort, corresponding to the values of the algebraic datatype, and 2) an Object sort, corresponding to all remaining values. In a programming language such as ML [34], we would define this algebraic datatype using a declaration such as:

```
datatype Tree = Nil | Node of Tree * Object * Tree
datatype Object = Obj1 | Obj2 | ... | ObjN
```

Note that the datatype Tree has an infinite set of values, because there is no bound on the size of a tree. On the other hand, we assume that we have already finitized the set Object.

Algebraic datatypes have proven to be useful not only in programming languages, but also in model theory, where they correspond to term algebras [29, Chapter 23], [14, Section 1.3]. Term algebras are algebras in which values are interpreted syntactically: given a term t without free variables (a *ground term*), the interpretation of t is t itself. The term algebras corresponding to the Tree datatype are generated by:

1) a constant Nil of sort Tree, and
2) a ternary constructor Node of type

$$\mathsf{Tree} \times \mathsf{Object} \times \mathsf{Tree} \to \mathsf{Tree}$$

A logic with transitive closure. We consider a fragment of first-order logic with transitive closure. The syntax of our logic is in Figure 1: the nonterminal S denotes set-valued expressions, R denotes relation-valued expressions, A denotes atomic formulas, B denotes quantifier-free formulas, and F denotes general (potentially quantified) formulas. The non-terminal S_V denotes sets (corresponding to one-place predicates), whereas the non-terminal R_V denotes binary relations (corresponding to two-place predicates). The term $\hat{}r$ denotes the transitive closure of the binary relation r, whereas $*r$ denotes the reflexive transitive closure of r. (Among the expressions that we intentionally omit are the universal set, relational transpose, and complement. These constructs make it difficult to ensure certain locality properties of the expressions that are useful for the formulation of our result.) We use the shorthand $\exists^1 x.F$ for the formula $\exists x.F(x) \wedge (\forall x, y.\ F(x) \wedge F(y) \Rightarrow x = y)$.

We interpret formulas in our logic over two-sorted structures $M = (T, O, \iota)$ where T is the domain of the sort Tree, O is the domain of the sort Object, and ι with domain $S_V \cup R_V$ interprets the built-in sets and relations of the structure M, so that $\iota(s) \subseteq T$ or $\iota(s) \subseteq O$ if s is a built-in set, and $\iota(r) \subseteq T^2$, $\iota(r) \subseteq O^2$, $\iota(r) \subseteq T \times O$, or $\iota(r) \subseteq O \times T$ if r is a built-in relation.

The standard model-theoretic semantics of our logic is in Figure 2. The function $\alpha : \mathsf{Vars} \to T \cup O$ is a valuation that maps each free scalar variable to its value. If φ is a sentence (a formula with no free variables) then $[\![\varphi]\!]^{M,\alpha}$ does not depend on α, so when φ is a sentence and $[\![\varphi]\!]^{M,\alpha} = \mathsf{true}$, we say that M is a *model* of the sentence φ. A structure M is a model of a set of sentences iff M is a model of each of the sentences in the set.

Note that, although we use set-theoretic notation such as $x \in S$ and $(x, y) \in R$, our logic does not allow quantification over sets, and is no stronger than first-order logic with transitive closure. Recall, however, that first-order logic is very expressive. Indeed, first-order logic has been used as a foundation for set theory and all of mathematics [33].

Axiomatizing term algebras in first order logic. Term algebras can be described using *constructor* relations, such as Node, or using *selector* relations, which are the inverse of the constructors [14, Section 2.6]. For our purpose, it is more convenient to use selectors. Because we consider a binary tree, we use the selectors left, content, and right, where left and right denote the children of a node in the tree, and content denotes the Object value associated with a tree node. We represent selectors as binary relations that are partial functions defined on non-Nil terms. We define the relation node as the following shorthand:

$$\begin{aligned}(t, t_1, o, t_2) \in \mathsf{node} \overset{\text{def}}{\Longleftrightarrow} {} & (t, t_1) \in \mathsf{left} \wedge (t, o) \in \mathsf{content} \wedge \\ & (t, t_2) \in \mathsf{right} \wedge\ t \neq \mathsf{Nil}\end{aligned}$$

$$
\begin{array}{rcl}
F & ::= & B \mid \forall x :: \mathsf{sort}.F \mid \exists x :: \mathsf{sort}.F \mid F_1 \wedge F_2 \mid \neg F_1 \\
B & ::= & A \mid B_1 \wedge B_2 \mid \neg B_1 \\
A & ::= & (x_1, x_2) \in R \mid x \in S \mid S_1 \subseteq S_2 \mid R_1 \subseteq R_2 \mid \\
 & & S_1 = S_2 \mid R_1 = R_2 \mid x_1 = x_2 \\
S & ::= & S_V \mid S_1 \,\mathsf{setOp}\, S_2 \mid S.R \mid \{x_1, \ldots, x_n\} \\
R & ::= & R_V \mid R_1 \,\mathsf{setOp}\, R_2 \mid R_1.R_2 \mid \Delta \mid {}^\frown R \mid *R \\
\mathsf{setOp} & ::= & \cup \mid \cap \mid \setminus \\
\mathsf{sort} & ::= & \mathsf{Tree} \mid \mathsf{Object}
\end{array}
$$

Figure 1: Syntax for a Logic with Transitive Closure

$$M = (T, O, \iota), \quad \alpha : \mathsf{Vars} \to T \cup O$$

$$
\begin{array}{rl}
[\![\forall x :: \mathsf{Tree}.F]\!]^{M,\alpha} = & \forall t \in T.\ [\![F]\!]^{M,\alpha'}, \quad \alpha' = \alpha[x := t] \\
[\![\forall x :: \mathsf{Object}.F]\!]^{M,\alpha} = & \forall o \in O.\ [\![F]\!]^{M,\alpha'}, \quad \alpha' = \alpha[x := o] \\
[\![\exists x :: \mathsf{Tree}.F]\!]^{M,\alpha} = & \exists t \in T.\ [\![F]\!]^{M,\alpha'}, \quad \alpha' = \alpha[x := t] \\
[\![\exists x :: \mathsf{Object}.F]\!]^{M,\alpha} = & \exists o \in O.\ [\![F]\!]^{M,\alpha'}, \quad \alpha' = \alpha[x := o] \\
[\![B_1 \wedge B_2]\!]^{M,\alpha} = & [\![B_1]\!]^{M,\alpha} \wedge [\![B_2]\!]^{M,\alpha} \\
[\![\neg B]\!]^{M,\alpha} = & \neg [\![B]\!]^{M,\alpha}
\end{array}
$$

$$
\begin{array}{rl}
[\![(x_1, x_2) \in R]\!]^{M,\alpha} = & (\alpha(x_1), \alpha(x_2)) \in [\![R]\!]^{M,\alpha} \\
[\![x \in S]\!]^{M,\alpha} = & \alpha(x) \in [\![S]\!]^{M,\alpha} \\
[\![S_1 \subseteq S_2]\!]^{M,\alpha} = & ([\![S_1]\!]^{M,\alpha} \subseteq [\![S_2]\!]^{M,\alpha}) \\
[\![R_1 \subseteq R_2]\!]^{M,\alpha} = & ([\![R_1]\!]^{M,\alpha} \subseteq [\![R_2]\!]^{M,\alpha}) \\
[\![S_1 = S_2]\!]^{M,\alpha} = & ([\![S_1]\!]^{M,\alpha} = [\![S_2]\!]^{M,\alpha}) \\
[\![R_1 = R_2]\!]^{M,\alpha} = & ([\![R_1]\!]^{M,\alpha} = [\![R_2]\!]^{M,\alpha}) \\
[\![x_1 = x_2]\!]^{M,\alpha} = & (\alpha(x_1) = \alpha(x_2))
\end{array}
$$

$$
\begin{array}{rl}
[\![S_1 \ \mathsf{setOp}\ S_2]\!]^{M,\alpha} = & [\![S_1]\!]^{M,\alpha} \ \mathsf{setOp}\ [\![S_2]\!]^{M,\alpha} \\
[\![S.R]\!]^{M,\alpha} = & \{y \mid \exists x \in [\![S]\!]^{M,\alpha}.\ (x, y) \in [\![R]\!]^{M,\alpha}\} \\
[\![\{x_1, \ldots, x_n\}]\!]^{M,\alpha} = & \{\alpha(x_1), \ldots, \alpha(x_n)\} \\
[\![R_1 \ \mathsf{setOp}\ R_2]\!]^{M,\alpha} = & [\![R_1]\!]^{M,\alpha} \ \mathsf{setOp}\ [\![R_2]\!]^{M,\alpha} \\
[\![R_1.R_2]\!]^{M,\alpha} = & [\![R_1]\!]^{M,\alpha} \circ [\![R_2]\!]^{M,\alpha} \\
 = & \{(x, z) \mid \exists y.\ (x, y) \in [\![R_1]\!]^{M,\alpha} \wedge \\
 & \qquad (y, z) \in [\![R_2]\!]^{M,\alpha}\} \\
[\![\Delta]\!]^{M,\alpha} = & \{(x, x) \mid x \in T\} \\
[\![{}^\frown R]\!]^{M,\alpha} = & \{(x_0, x_n) \mid \exists n \geq 1. \exists x_1, \ldots, x_{n-1} \in T. \\
 & \quad \bigwedge_{i=1}^{n} (x_{i-1}, x_i) \in [\![R]\!]^{M,\alpha}\} \\
[\![*R]\!]^{M,\alpha} = & [\![\Delta]\!]^{M,\alpha} \cup [\![{}^\frown R]\!]^{M,\alpha} \\
[\![S_V]\!]^{M,\alpha} = & \iota(S_V) \\
[\![R_V]\!]^{M,\alpha} = & \iota(R_V)
\end{array}
$$

Figure 2: Semantics for Logic of Figure 1

We also use the subterm relation, defined using transitive closure:

$$\mathsf{subterm} \overset{\text{def}}{=} {}^\frown(\mathsf{left} \cup \mathsf{right})$$

The term model. We are interested in checking the satisfiability of formulas over the term model $M_T = (T_T, O, \iota_T)$ given as follows:

- T_T is the set of ground terms generated by the constant Nil and the constructor Node; in other words, T is the least set such that
  1. $\mathsf{Nil} \in T_T$, and
  2. if $t_1, t_2 \in T_T$ and $o \in O$, then $\mathsf{Node}(t_1, o, t_2) \in T_T$.
- O is a finite set;
- ι_T is defined as follows:

$$\iota_T(\mathsf{Nil}) = \mathsf{Nil}$$
$$\iota_T(\mathsf{left}) = \{(\mathsf{Node}(t_1, o, t_2), t_1) \mid t_1, t_2 \in T_T, o \in O\}$$
$$\iota_T(\mathsf{content}) = \{(\mathsf{Node}(t_1, o, t_2), o) \mid t_1, t_2 \in T_T, o \in O\}$$
$$\iota_T(\mathsf{right}) = \{(\mathsf{Node}(t_1, o, t_2), t_2) \mid t_1, t_2 \in T_T, o \in O\}$$

Figure 3 sketches one part of the structure M_T.

Axioms for term algebras. We adopt the following axioms to describe the properties of term algebras.

Selectors: The binary relations left, content, and right are total functions on the non-Nil elements of the sort Tree, and are undefined on Nil:

1. $\forall t :: \mathsf{Tree}.\ t \neq \mathsf{Nil} \Rightarrow (\exists^1 t_1 :: \mathsf{Tree}.\ (t, t_1) \in \mathsf{left}) \wedge$ $(\exists^1 o :: \mathsf{Object}.\ (t, o) \in \mathsf{content}) \wedge$ $(\exists^1 t_2 :: \mathsf{Tree}.\ (t, t_2) \in \mathsf{right})$
2. $\forall t :: \mathsf{Tree}. \forall o :: \mathsf{Object}.\ (\mathsf{Nil}, t) \notin \mathsf{left} \wedge$ $(\mathsf{Nil}, o) \notin \mathsf{content} \wedge$ $(\mathsf{Nil}, t) \notin \mathsf{right}$

We assume that a simple type system of our two-sorted language rules out the application of relations to elements of inappropriate sort; for example, if $t :: \mathsf{Tree}$ and $o :: \mathsf{Object}$, then $(t, o) \in \mathsf{left}$ is not a well-formed formula.

Uniqueness: The defined relation node has the properties of a partial function:

$$\forall t, t', t_1, t_2 :: \mathsf{Tree}. \forall o :: \mathsf{Object}.$$
$$(t, t_1, o, t_2) \in \mathsf{node} \wedge (t', t_1, o, t_2) \in \mathsf{node} \Rightarrow t = t'$$

Generator: The defined relation node has the properties of a total function:

$$\forall t_1, t_2 :: \mathsf{Tree}. \forall o :: \mathsf{Object}.\ \exists t :: \mathsf{Tree}.\ (t, t_1, o, t_2) \in \mathsf{node}$$

This axiom holds in M_T, but we will consider the consequences of omitting it from the axiomatization.

Acyclicity: A term is never a proper subterm of itself; that is, the subterm relation is acyclic:

$$\forall t :: \mathsf{Tree}.\ (t, t) \notin \mathsf{subterm}$$

We denote by SUGA the conjunction of the axioms above (taking the first letter of the name of each axiom).

Note that the SUGA axioms have no finite models. Namely, although not all models of SUGA are isomorphic, they all contain an infinite chain of elements $t_0, t_1, t_2, \ldots$ where $t_0 = \mathsf{Nil}$ and $(t_{i+1}, t_i, o, \mathsf{Nil}) \in \mathsf{node}$. These elements exist by the *Generator* axiom; the *Acyclicity* axiom guarantees that they are all distinct because they are ordered by the subterm relation.

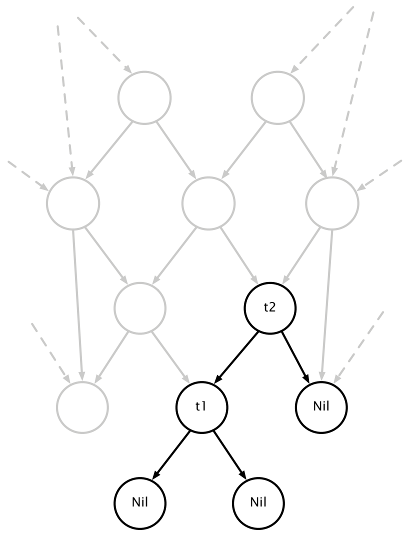

Figure 3: Sketch of the infinite structure M_T and an example of one of its finite subterm-closed substructures $M_0 = (T_0, \{o\}, \iota_0)$ where $T_0 = \{\mathsf{Nil}, t_1, t_2\}$ for $t_1 = \mathsf{Node}(\mathsf{Nil}, o, \mathsf{Nil})$ and $t_2 = \mathsf{Node}(t_1, o, \mathsf{Nil})$. The edges corresponding to the content relation are omitted for clarity.

4. FINITE SATISFIABILITY RESULT

This section presents the main results of our paper, which enable the checking of properties of algebraic datatypes using finite models. The basic idea of our approach is the following: to prevent all models from being infinite, we drop the *Generator* axiom. Denote by SUA the conjunction of the remaining axioms (*Selectors*, *Uniqueness*, *Acyclicity*). It turns out that, among the finite structures, SUA characterizes precisely the substructures of the term model that are subterm-closed (if t is in the structure, then so is each subterm of t). Having established this, we identify a class of sentences whose validity in a finite model implies their validity in the full infinite model M_T.

We next define the notion of a subterm-closed finite substructure of M_T, illustrated in Figure 3. Intuitively, a finite substructure $M_0 = (T_0, O, \iota)$ of M_T is a structure obtained from M_T by selecting a finite set T_0 of terms and preserving all the relations between the terms in T_0. A structure is subterm-closed if every subterm of a term in T_0 is also in T_0.

More precisely, let $M_T = (T_T, O, \iota_T)$ be the term model. A *substructure of* M_T is a structure $M_0 = (T_0, O, \iota_0)$ where $T_0 \subseteq T_T$ and the relations given by ι_0 are restrictions of the corresponding relations given by ι_T, that is, $\iota_0(\mathsf{left}) = \iota_T(\mathsf{left}) \cap T_0^2$, $\iota_0(\mathsf{right}) = \iota_T(\mathsf{right}) \cap T_0^2$, and $\iota_0(\mathsf{content}) = \iota_T(\mathsf{content}) \cap T_0 \times O$. (We have for simplicity assumed that substructures have the same domain O of values of the sort Object.) A *subterm-closed finite substructure of* M_T is a finite substructure M_0 of M_T whose domain of terms T_0 satisfies the property

$$t \in T_0 \ \wedge \ (t, t_1) \in [\![\mathsf{subterm}]\!]^{M_T, \alpha} \Rightarrow t_1 \in T_0$$

for all $t, t_1 \in T$.

We then have the following completeness theorem that explains why the SUA axioms are adequate. This theorem allows us to ensure that any model of SUA axioms has precisely the properties of a subterm-closed finite substructure of the term model M_T. The proof of Theorem 1 is in the Appendix.

Theorem 1 (Axiomatization of Finite Substructures) *Let $M = (T, O, \iota)$ be a finite two-sorted structure with language $\langle \mathsf{Nil}, \mathsf{left}, \mathsf{content}, \mathsf{right} \rangle$. Then M is a model of* SUA *iff M is isomorphic to some subterm-closed finite substructure M_0 of M_T.*

Having identified that the SUA axioms enforce that a finite structure "looks like" a term algebra M_T, we turn to the question of when checking a property in finite models is sufficient to ensure that the property holds in the full, infinite model M_T. We first look at sentences that contain only existential quantifiers.

An *existential sentence* φ is a formula of the form

$$\exists t_1 :: \mathsf{Tree}. \ \ldots \ \exists t_n :: \mathsf{Tree}. \ \psi$$

where ψ is quantifier-free and the free variables of ψ are $t_1, \ldots, t_n$. The following is a fundamental property of structures (dual to the property that if a universal sentence holds in a structure then it also holds in its substructure).

Fact 1 *Let M_0 be a substructure of M and φ an existential sentence. If φ holds in M_0, then φ also holds in M.*

Informally, the property holds because the same existential witnesses from the smaller structure can be used in the larger structure.

An important consequence of Theorem 1 and Fact 1 is the following: if φ is an existential sentence and the conjunction $\mathsf{SUA} \wedge \varphi$ holds in a finite two-sorted model M_1, then φ holds in the full term model M_T. Indeed, if $\mathsf{SUA} \wedge \varphi$ holds in M_1, because SUA holds in φ, by Theorem 1, M_1 is isomorphic to some subterm-closed finite substructure M_0, so φ holds in M_0 as well. By Fact 1, φ holds in M_T as well. We thus obtain a method to check whether a formula φ holds in M_T: check φ in each finite subterm-closed model of M_T. In the rest of this section, we generalize this result by allowing arbitrary quantification in φ, as long as it is bounded by previously introduced values. The intuitive reason why this generalization is possible is that we are checking formulas on subterm-closed structures, which means that the bounded quantification has the same semantics in the substructure M_0 and in the full infinite structure M_T.

Let S be a set-valued term, denoted S in Figure 1, that does not contain variable t. A *bounded universal term quantifier*

$$\forall_S \, t :: \mathsf{Tree}. \, F$$

is a shorthand for the formula $\forall t :: \mathsf{Tree}. \ t \in S \ \Rightarrow \ F$. Dually, a *bounded existential term quantifier*

$$\exists_S \, t :: \mathsf{Tree}. \, F$$

is a shorthand for the formula $\exists t :: \mathsf{Tree}. \ t \in S \ \wedge \ F$.

Hence, bounded quantifiers are expressible in terms of the ordinary quantifiers, but are more restrictive. An *existential—bounded-universal sentence* requires each universal quantifier to be bounded by some set expression S. More precisely, we have the following definition:

Definition 1 *An* existential—bounded-universal sentence *(an* EBU *sentence) is a formula of the form*

$$Q_1 v_1 :: s_1. \ \ldots \ Q_n v_n :: s_n. \ \psi$$

where ψ is a quantifier-free formula (denoted B in Figure 1) and each $Q_i v_i :: s_i$ is a quantifier or a bounded quantifier of one of the following forms:

- *An existential term quantifier $\exists v_k :: \mathsf{Tree}$;*

```
procedure TermSat
input:  $\varphi$: an EBU sentence
output: if $\varphi$ is true in the term model $M_T$:
          a finite substructure $M_0$ of $M_T$ where $\varphi$ holds
        if $\varphi$ is false in the term model $M_T$:
          no result (infinite loop)
begin
  k := 1;
  while (true) do
    for each model $M_1 = (T, O, \iota)$ with $|T| + |O| = k$ do
      if $(\mathsf{SUA} \wedge \varphi)$ holds in $M_1$ then return $M_1$; fi
    end
    k := k + 1;
  end
end TermSat.
```

Figure 4: A semidecision procedure for checking satisfiability of EBU sentences in the term model M_T.

- *A bounded universal term quantifier* $\forall_S v_k$ :: Tree *where the free variables of the set-valued term S are among the previously quantified variables $v_1, \ldots, v_{k-1}$;*
- *A universal object quantifier* $\forall v_k$ :: Object*;*
- *An existential object quantifier* $\exists v_k$ :: Object.

We write $\varphi \in$ EBU to denote that φ is an EBU sentence.

Note the ways in which EBU sentences generalize purely existential sentences: not only is it possible to have arbitrary bounded quantifiers, it is also possible to introduce new unrestricted existential quantifiers, even after bounded universal quantifiers.

We are now ready to state our main theorem. The proof of Theorem 2 is in the Appendix.

Theorem 2 (Finite Satisfiability Theorem) *Let φ be an EBU sentence and M_T a term model. Then φ holds in M_T iff it holds in some subterm-closed finite substructure M_0 of M_T.*

5. CONSEQUENCES

Given the results of the previous section, we can now answer the question we posed in Section 2. An analysis of an Alloy model in which algebraic datatypes are encoded relationally will yield sound counterexamples so long as the formula being checked is in EBU. A syntactic check for membership in EBU based on Definition 1 is easy to implement. The result extends to bounded model checkers (such as NuSMV [6]) whose analysis consists of finding a model of a formula. The language of formulas for such model checkers could be soundly extended to EBU sentences over algebraic datatypes.

Analysis procedure. The analysis procedure suggested by our result is shown in Figure 4: to check whether an EBU sentence φ holds in M_T, search for increasingly large finite models of SUA$\wedge\varphi$. The procedure captures the spirit of analyses such as that performed by the Alloy Analyzer [1,17]; in practice, the search for models would employ pruning and heuristics and would not require an exhaustive enumeration. The correctness and completeness of the procedure follows from the results of this section and is the main result of this paper:

- we can check the condition that a structure is a subterm-closed submodel of M_T by simply conjoining axioms SUA to φ, thanks to Theorem 1;
- we know that the existence of the returned finite model M_1 implies that φ holds in M_T, thanks to the soundness ($\Longleftarrow$) direction of Theorem 2;
- we know that if φ holds in model M_T, then the algorithm will find a finite model M_1 which proves this fact, thanks to the completeness ($\Longrightarrow$) direction of Theorem 2.

Closure under boolean operations. Having identified EBU sentences as a useful class of formulas for which the algorithm in Figure 4 is applicable, we examine the following question. If $\varphi_1, \varphi_2 \in$ EBU, is there an effectively constructible sentence $\varphi \in$ EBU such that the following equivalences hold in M_T?

- $\varphi \iff (\varphi_1 \wedge \varphi_2)$
- $\varphi \iff (\varphi_1 \vee \varphi_2)$
- $\varphi \iff (\neg\varphi_1)$
- $\varphi \iff (\varphi_1 \Rightarrow \varphi_2)$

It turns out that the answer to first two questions is "yes", whereas the answer to the last two questions is "no". In other words, EBU sentences are closed only under positive boolean combinations, but are not closed under negation or implication. We make this claim precise in three propositions that follow.

Proposition 1 *Let $\varphi_1 \equiv \mathrm{BQ}_1.F_1$ and $\varphi_2 \equiv \mathrm{BQ}_2.F_2$ be EBU sentences where BQ_1 and BQ_2 denote sequences of quantifiers and bounded quantifiers and where F_1, F_2 are quantifier-free formulas. Let $\mathrm{BQ}'_2.F'_2$ be the result of renaming the variables in φ_2 so that they are all distinct from the variables in φ_1. Then*

$$\begin{aligned} \varphi_1 \wedge \varphi_2 &\iff \mathrm{BQ}_1.\mathrm{BQ}'_2.\,F_1 \wedge F'_2 \\ \varphi_1 \vee \varphi_2 &\iff \mathrm{BQ}_1.\mathrm{BQ}'_2.\,F_1 \vee F'_2 \end{aligned} \tag{1}$$

Moreover, $\mathrm{BQ}_1.\mathrm{BQ}'_2.F_1 \wedge F'_2$ and $\mathrm{BQ}_1.\mathrm{BQ}'_2.F_1 \vee F'_2$ are EBU sentences.

The condition (1) follows from the basic monotonicity property of quantifiers and operations $\wedge, \vee$. The fact that the concatenation of disjoint EBU sequences of quantifiers is again an EBU sequence of quantifiers follows from the definition of EBU sentences.

We next turn to the absence of the closure under negation and implication. We first note that the entire class of EBU sentences is undecidable.

Fact 2 *The problem of determining, given an EBU sentence φ, whether φ holds in M_T, is undecidable.*

Fact 2 follows from the undecidability result in [42, Section 4], which shows a reduction from the Post correspondence problem to the satisfiability of term algebra sentences with subterm relation, existential quantifiers, and bounded universal quantifiers.

Fact 2 has two main consequences for this paper. The first consequence is that, from the viewpoint of computability, the semidecision procedure in Figure 4 is as good as we can hope for. The second consequence is the absence of closure under negation and implication, as given by the following propositions.

Proposition 2 *There is no algorithm that, given a sentence $\varphi \in$ EBU, constructs an EBU sentence equivalent to $\neg\varphi$.*

Proof. The proof is by contradiction. Suppose that there is such an algorithm. Consider any EBU sentence φ. Let $\bar{\varphi}$ be the EBU sentence computed by the algorithm, so that $\bar{\varphi}$ is equivalent to $\neg\varphi$. Then either φ or $\bar{\varphi}$ holds in M_T, so if we run two copies of the procedure in Figure 4 in parallel, one with the input φ and the other one with the input $\bar{\varphi}$, then one of the algorithms will eventually terminate and we will conclude that φ is either true or false. This implies that the class of EBU formulas is decidable, contradicting Fact 2. ∎

Because $\neg\phi$ is equivalent to $(\phi \Rightarrow \mathsf{false})$, we obtain the absence of closure under negation as well.

Proposition 3 *There is no algorithm that, given sentences φ and ψ constructs an* EBU *sentence equivalent to $\varphi \Rightarrow \psi$.*

Arbitrary algebraic datatypes. We have presented our result for binary trees, but it applies to all algebraic datatypes, and, more generally, to any structured data. Indeed, it is clearly possible in relational logic to reason about records and tuples that have an *a priori* bounded number of components: just introduce a new variable for each component. What the results of this paper imply is that it is also possible to reason about structures, such as binary trees, that do not have an *a priori* bound on their size. It is not difficult to generalize the proofs of Theorem 1 and Theorem 2 to the case of any finite number of mutually recursive algebraic datatypes. Alternatively, we can encode any number of datatypes using binary trees. (Indeed, the experience with programming languages such as LISP [31] is convincing evidence that data structures can be represented using LISP-like lists, which are binary trees.) The idea of representing algebraic datatypes with trees is to replace each constructor application $C_k(t_1, \ldots, t_n, o_1, \ldots, o_m)$ with an expression

$$\begin{array}{l}\mathsf{Node}(f_k, o_0,\\ \quad \mathsf{Node}(t_1, o_1, \mathsf{Node}(t_2, o_2, \ldots \mathsf{Node}(t_P, o_P, \mathsf{Nil}) \ldots)))\end{array}$$

where f_k is a finite tree (of size $O(\log k)$) that encodes the name of the constructor C_k, where $P = \max(n, m)$, $t_i = \mathsf{Nil}$ if $i > n$, and $o_i = o_0$ if $i > m$. Here $o_0 \in O$ is some arbitrary fixed object from the set of uninterpreted objects. The corresponding selector relations are similarly definable using quantifier-free formulas in terms of selectors left and right, and so is the subterm relation. Note that, when reasoning about arbitrary algebraic datatypes, we are interested not in all possible trees, but only in the substructure of M_T which is the image of the embedding. In other words, we would like to ensure that the binary trees that represent the values of variables in formulas are consistent with the type system of the original algebraic datatypes. We can express this condition by quantifying only over the terms whose every subterm respects the local structure given by the original type system; this condition is expressible using our logic with transitive closure. Therefore, it suffices to restrict all quantified variables to the terms that satisfy this condition, and the resulting formula can be checked using the algorithm in Figure 4.

The scope of our result. After realizing that our technique applies to algebraic datatypes, a natural question to ask is: does the technique fundamentally depend on the properties of the structure of algebraic datatypes, such as the uniqueness of left and right relations (as given by the *Selectors* axiom), the uniqueness of the parent relation node (as given by the *Uniqueness* axiom), or even the acyclicity (given by the *Acyclicity* axiom)? When examining this question it is worthwhile to consider two separate questions:

- How do we generalize the notion of subterm-bounded substructures of M_T to the case of substructures of some other infinite structure M_∞ of interest? (The generalization of Theorem 2.)

  Suppose that we are interested in checking constraints over an infinite structure M_∞ with relation symbols $r_1, \ldots, r_n$. It turns out that the only essential requirement on the structure M_∞ is that, for some term variable t, the set $[\![\{t\}.{*}(r_1 \cup \ldots \cup r_n)]\!]^{M_\infty, \alpha}$ is finite for each valuation α. In other words, as long as the set of elements "below" each element of M_∞ is finite, we can use bounded quantification to reduce the truth value of EBU sentences in M_∞ to the satisfiability in finite substructures closed under the "below" relation. In particular, the technique applies to structures that contain shared elements and cycles.

- How do we axiomatize a class of finite structures of interest? (The generalization of Theorem 1.)

  From an algorithmic point of view, this question admits a wider spectrum of solutions than just the use of axioms in first-order logic with transitive closure (although the use of axioms may have an advantage in the context of constraint-solving tools). Indeed, given a family of finite structures of interest (in particular, given a family of finite subterm-closed substructures of M_∞) we can use any language of computable functions to define an executable test predicate that determines whether a finite structure is isomorphic to one of the finite structures of interest. In other words, we can use an algorithm specialized for a given problem to filter the finite structures of interest. This idea of using "executable predicates" appears in the form of runtime assertions in many programming languages and has found applications in software testing [5, 30].

Because of these generalizations, we expect our result to be applicable to a range of infinite structures.

Despite our characterization of EBU, two practical problems remain for a modelling language such as Alloy. The first, and simpler, problem is how the language might exploit our result. The second is that some important modelling constraints seem inexpressible in EBU.

By design, Alloy has no unbounded quantification, but the use of a signature as the bounding expression gives the desired semantic effect. Not all signatures in a model—in fact, very few in practice—represent algebraic datatypes, so the generator axiom should not be implicit for all signatures. Perhaps a signature declaration could be labelled with a special keyword thus:

```
datatype sig List {...}
```

The analyzer would then identify non-EBU formulas in which a variable is universally quantified over such a signature, and give an appropriate warning.

In our experience, non-EBU formulas are rarely needed. They arise mainly for novices, who have not yet assimilated the relational idiom, and try to use an algebraic structure when a simpler and cleaner relational structure would suffice. More expert users have, however, encountered the problem when using sequences. A model of network routing, for example, had a node add a path to its routing table only when shorter than the concatenation of other paths that could be computed; this concatenation implicitly involved a non-EBU formula.

It is actually more common for the problem of an undesirable generator axiom to arise for the degenerate case of a non-recursive datatype. The states of a system, for example, are usually described as a Cartesian product of components. When using an explicit element to represent each state, the generator axiom would require the existence of a state for every combination of component values. Here, the problem is not that the axiom would require an infinite model, but that it would require one too large to analyze in practice. In the absence of this generator axiom on the states of the system, assertions about preconditions of operations, for example, produce spurious counterexamples. The tractability of the constraint solving analysis depends on considering only those models with enough states to refute an assertion; for an assertion about invariant preservation, for example, this means just two states: the pre- and post-states. But with the generator axiom, any model must include all possible states. Whether these problems can be solved remains to be seen.

6. RELATED WORK

Constraint-checking tools. Because of its full automation, model checking approaches based on finitization of the problem space are very attractive. These approaches have had great success for control-intensive problems [6, 20] such as those arising in hardware verification. The complexity of software systems often comes from the data structures that they manipulate, and notations such as UML [38] have been used to describe such constraints. The Alloy notation [17,19] can also be used to describe such constraints; the Alloy Analyzer tool [1] can then search for the structures that satisfy these constraints. Our experience in using the Alloy notation and the analyzer to reason about structured values was the immediate inspiration for this paper. Because it establishes a general correspondence between satisfiability in finite and infinite models, our result is potentially applicable not only to Alloy, but also to tools such as MACE [32], Paradox [7], USE [10], ProB [26], RACER [13], and FaCT [15].

Algebraic datatypes. Our paper uses algebraic datatypes as a well-studied example of unbounded structured values. Algebraic datatypes are the basis of the algebraic approach to formal specification and verification [2, 4, 11, 12]. The use of the list algebraic datatype was pioneered by LISP [31]. User-defined algebraic datatypes go back to ML [34] and are used in Objective Caml [25] and Haskell [3].

Term algebras without transitive closure. The first-order theory of term algebras is decidable [28, 29, 40]. Because the interpretation of Object is a finite set, omitting the transitive closure from our logic makes formulas decidable even with arbitrary (not only bounded) quantifiers. The complexity of the resulting decision problem is non-elementary [8, 9] with the height of the tower of exponentials linear in the number of quantifier alternations in the formula [44]. More tractable classes of term algebras include the class of quantifier-free formulas [36]. Several extensions of term algebras have been proved decidable [23, 24, 41, 43], mostly using quantifier elimination techniques.

Term algebras with a subterm relation. Adding a subterm relation to the first-order language of term algebras makes the problem substantially more difficult. Indeed, [42] shows that even the satisfiability of formulas with bounded universal quantifiers is undecidable (although the satisfiability of the purely existential fragment with a subterm relation is still decidable). As we noted in Section 5, the undecidability result for term algebras with subterms applies to our logic as well, because the subterm relation is expressible using transitive closure. A search for counterexamples is useful even for an undecidable logic (and is, in fact, at least as important as the search for counterexamples in decidable logics), and the results of this paper show how to perform such search for a useful class of formulas.

First-order logic with transitive closure. First-order logic with transitive closure is useful for reasoning about program data structures and has been used not only in Alloy [19], but also in shape analysis tools such as TVLA [27] and PALE [35]. Among the decidable fragments with transitive closure are monadic second-order logic [21,37] and some subclasses of the existential monadic second-order logic of graphs [16].

7. CONCLUSIONS

The language of sets and relations has proven to be a powerful notation for modelling a range of structures arising in software design and analysis. Model finding tools have made this approach accessible and practical. So far, model finding tools have been restricted to arbitrarily large, but finite models. However, some useful structures are inherently infinite, in particular the algebraic datatypes such as lists and trees. When we try to apply existing tools to these structures, we are faced, in general, with either ruling out all models (which is sound, but entirely useless), or allowing the possibility that the tool returns unsound, meaningless models that do not apply to the desired infinite structures.

We have presented a useful and natural class of formulas for which the existence of a finite model reveals the satisfiability of the formula in the infinite structure. For this class of properties, we have proved that it is possible to partially axiomatize the desired structure in such a way that finite models are simply substructures of the desired infinite structure. In this way, concrete feedback from model finding tools can be brought to a range of ubiquitous data structures that would otherwise remain out of their scope.

8. REFERENCES

[1] The Alloy project. http://alloy.mit.edu/.

[2] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.

[3] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, Inc., 2nd edition, 1998.

[4] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1-2):35–132, 2000.

[5] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, July 2002.

[6] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in LNCS, pages 495–499, Trento, Italy, July 1999. Springer.

[7] K. Claessen and N. Sörensson. New techniques that improve MACE-style model finding. In *Model Computation*, 2003.

[8] K. J. Compton and C. W. Henson. A uniform method for proving lower bounds on the computational complexity of logical theories. *Annals of Pure and Applied Logic*, 48(1):1–79, July 1990.

[9] J. Ferrante and C. W. Rackoff. *The Computational Complexity of Logical Theories*, volume 718 of *Lecture Notes in Mathematics*. Springer-Verlag, 1979.

[10] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Journal on Software and System Modeling*, 2005. to appear.

[11] J. V. Guttag. Abstract data types, then and now. In *Software Pioneers: Contributions to Software Engineering*, pages 442–452. Springer, 2002.

[12] J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, 1978.

[13] V. Haarslev and R. Möller. RACER system description. In *International Joint Conference on Automated Reasoning*, 2001.

[14] W. Hodges. *Model Theory*, volume 42 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1993.

[15] I. Horrocks. Using an expressive description logic: FaCT or fiction? In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 636–647, 1998.

[16] N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Computer Science Logic (CSL)*, pages 160–174, 2004.

[17] D. Jackson. Automating first-order relational logic. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, 2000.

[18] D. Jackson. Object models as heap invariants. In A. McIver and C. Morgan, editors, *Collected Papers of IFIP Working Group 2.3 on Programming Methodology*. Springer-Verlag, 2001.

[19] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering / European Software Engineering Conference (FSE/ESEC '01)*, 2001.

[20] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[21] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. In *Proc. 5th International Conference on Implementation and Application of Automata*. LNCS, 2000.

[22] V. Kuncak and D. Jackson. On relational analysis of algebraic datatypes. Technical Report 985, MIT, April 2005.

[23] V. Kuncak and M. Rinard. On the theory of structural subtyping. Technical Report 879, Laboratory for Computer Science, Massachusetts Institute of Technology, 2003.

[24] V. Kuncak and M. Rinard. Structural subtyping of non-recursive types is decidable. In *Eighteenth Annual IEEE Symposium on Logic in Computer Science*, 2003.

[25] X. Leroy. *The Objective Caml system, release 3.08*, July 2004.

[26] M. Leuschel and M. J. Butler. ProB: A model checker for B. In *Formal Methods Europe*, pages 855–874, 2003.

[27] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Proc. 7th International Static Analysis Symposium*, 2000.

[28] M. J. Maher. Complete axiomatizations of the algebras of the finite, rational, and infinite trees. *IEEE Symposium on Logic in Computer Science*, 1988.

[29] A. I. Mal'cev. *The Metamathematics of Algebraic Systems*, volume 66 of *Studies in Logic and The Foundations of Mathematics*. North Holland, 1971.

[30] D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, MIT, 2005.

[31] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part 1. *Comm. A.C.M.*, 3:184–195, 1960.

[32] W. McCune. MACE 2.0 Reference Manual and Guide. *ArXiv Computer Science e-prints*, June 2001.

[33] E. Mendelson. *Introduction to Mathematical Logic*. Chapman & Hall, London, 4th edition, 1997.

[34] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Mass., 1997.

[35] A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Programming Language Design and Implementation*, 2001.

[36] D. C. Oppen. Reasoning about recursively defined data structures. *Journal of the ACM*, 27(3), 1980.

[37] M. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.

[38] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, Reading, Mass., 1999.

[39] J. R. Rumbaugh, M. R. Blaha, W. Lorensen, F. Eddy, and W. Premerlani. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey 07632, USA, 1991.

[40] T. Sturm and V. Weispfenning. Quantifier elimination in term algebras: The case of finite languages. In V. G. Ganzha, E. W. Mayr, and E. V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing (CASC)*, TUM Muenchen, 2002.

[41] R. Treinen. Feature trees over arbitrary structures. In P. Blackburn and M. de Rijke, editors, *Specifying Syntactic Structures*, chapter 7, pages 185–211. CSLI Publications and FoLLI, 1997.

[42] K. N. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *Journal of the ACM (JACM)*, 34(2):492–510, 1987.

[43] T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for recursive data structures with integer constraints. In *International Joint Conference on Automated Reasoning*, volume 3097 of *LNCS*, pages 157–167, 2004.

[44] T. Zhang, H. B. Sipma, and Z. Manna. Term algebras with length function and bounded quantifier alternation. In *Theorem Proving in Higher-Order Logics*, volume 3223 of *LNCS*, pages 321–336, 2004.

APPENDIX

Proofs of Theorems

Theorem 1 (Axiomatization of Finite Substructures) *Let $M = (T, O, \iota)$ be a finite two-sorted structure with language $\langle \mathsf{Nil}, \mathsf{left}, \mathsf{content}, \mathsf{right}\rangle$. Then M is a model of* SUA *iff M is isomorphic to some subterm-closed finite substructure M_0 of M_T.*

Proof. We prove both directions of the equivalence.

$\Longleftarrow$): Suppose that a structure M is isomorphic to a subterm-closed model M_0 of M_T. Then M satisfies the same formulas as M_0. Therefore, it suffices to verify that M_0 satisfies the SUA axioms *Selectors*, *Uniqueness*, *Acyclicity*. Axioms *Uniqueness* and *Acyclicity* hold in M_T, so they hold in M_0 as well. Indeed, taking the contrapositive, a relation that has two values in a substructure M_0 also has two values in the larger structure M_T; and a cycle in M_0 is also a cycle in M_T. Axiom *Selectors* holds because M_0 is subterm-closed: the components of every non-Nil term t in T_0 are also in T_0.

$\Longrightarrow$): Suppose that a finite two-sorted structure $M = (T, O, \iota)$ satisfies the SUA axioms. We identify a subterm-closed finite structure $M_0 = (T_0, O, \iota_0)$ isomorphic to M, by establishing a relation $f \subseteq T \times T_T$ between the T elements that we hope to behave as terms (because they satisfy the SUA axioms), and the elements T_T of the term model M_T. We then let $g = f \cup \Delta_O$ where Δ_O is the identity relation on O, and show that g is a partial isomorphism. We also show that the domain of g is the entire domain $T \cup O$ of M. Therefore, M is isomorphic to the finite substructure M_0 of M_T induced by the range of g.

To define f, we start by mapping $\iota(\mathsf{Nil})$ to the element Nil from T_T, and extend f by following the parent relation in both M and M_0. Formally, we define f using a least fixpoint construction. Let $f_0 = \{(\iota(\mathsf{Nil}), \mathsf{Nil})\}$ and let

$$\begin{aligned} f_{i+1} = f_i \cup \{(t', \mathsf{Node}(t_1, o, t_2)) \in f \mid \exists t_1', t_2'.\ & \\ (t', t_1') \in \iota(\mathsf{left}), (t', t_2') \in \iota(\mathsf{right}), & \\ (t', o) \in \iota(\mathsf{content}), & \\ (t_1', t_1), (t_2', t_2) \in f_i\} & \end{aligned}$$

Define $f = \cup_{i \geq 0} f_i$. We show by induction that f is a partial isomorphism whose domain is T. To make sure that we have taken into account all elements of T, we define a measure d on the elements of structures M and M_T. Consider first an element $t \in T$ of structure M and consider a sequence of elements $t_0, t_1, \ldots$ such that $t_0 = t$ and $(t_i, t_{i+1}) \in \iota(\mathsf{left}) \cup \iota(\mathsf{right})$. Because M satisfies *Acyclicity* and T is finite, each such sequence is finite, and there is a finite number of such sequences. For each element t, let $d(t)$ be the maximum of the lengths of all such sequences. We analogously define $d(t)$ for $t \in T_T$.

Given this setup, we can easily prove by induction on i the conjunction of the following properties:

P1) $\mathsf{dom}(f_i) = \{t' \in T \mid d(t') \leq i\}$

P2) each relation $g_i = f_i \cup \Delta_O$ is a partial isomorphism, that is, that g_i is an isomorphism between structures induced by the domain of g_i (denoted $\mathsf{dom}(g_i)$) and the range of g_i (denoted $\mathsf{ran}(g_i)$);

The details of the inductive proof are given in the accompanying technical report [22].

Given that these properties hold for all i, let $n = \max\{d(t') \mid t' \in T\}$. Then $\mathsf{dom}(f_n) = T$ and $g_{f+1} = f_n$, so $g = g_n$. Let $M_0 = (T_0, O, \iota_0)$ where $T_0 = \mathsf{ran}(f_n)$ and $\iota_0(\mathsf{content}) = \{(t, o) \mid \exists t' \in T. (t', o) \in \iota(\mathsf{content})\}$. Then g is a bijection $T \cup O \to T_0 \cup O$, it preserves left and right because f_n does, and it preserves content by construction.

Therefore, M_0 is the desired finite substructure of M_T, and g is the desired isomorphism between M and M_0, which proves our claim. ■

Theorem 2 (Finite Satisfiability Theorem) *Let φ be an* EBU *sentence and M_T a term model. Then φ holds in M_T iff it holds in some subterm-closed finite substructure M_0 of M_T.*

Proof. We prove both directions of the equivalence.

Soundness ($\Longleftarrow$): Suppose that φ holds in a subterm-closed finite substructure $M_0 = (T_0, O, \iota_0)$ of $M_T = (T_T, O, \iota_T)$. When evaluating φ in M_T, for any witness for an existential quantifier we can pick the same witness in M_T as in M_0, because $T_0 \subseteq T_T$. Moreover, regardless of whether they are interpreted in M_0 or M_T, the universal quantifiers range only over elements of T_0, so they still hold in M_0. We next make this argument more precise.

Observe the following properties of set-valued and relation-valued terms in our language, for every structure M and every valuation α:

- if R is a relation-valued expression, then
$$[\![R]\!]^{M,\alpha} \subseteq [\![*(\mathsf{left} \cup \mathsf{right})]\!]^{M,\alpha} \tag{2}$$
- if S is a set-valued term with free variables $x_1, \ldots, x_n$ on which α is defined, then
$$[\![S]\!]^{M,\alpha} \subseteq [\![\{x_1, \ldots, x_n\}.*(\mathsf{left} \cup \mathsf{right})]\!]^{M,\alpha} \tag{3}$$

These properties follow by induction on the size of the expressions R and S.

Note also that M_0 is a substructure of M_T, so by induction on the size of R and S we have
$$\begin{array}{l} [\![R]\!]^{M_0,\alpha} = [\![R]\!]^{M_T,\alpha} \cap T_0^2 \\ [\![S]\!]^{M_0,\alpha} = [\![S]\!]^{M_T,\alpha} \cap T_0^2 \end{array} \tag{4}$$
for $\alpha : \mathsf{Vars} \to T_0 \cup O$.

We next show that the truth-value of a quantifier-free formula F is the same in M_0 and M_T when the free variables of F are interpreted in T_0. We show by induction on the structure of formula F that:

For all $\alpha : \mathsf{Vars} \to T_0 \cup O$,
$$[\![F]\!]^{M_0,\alpha} = [\![F]\!]^{M_T,\alpha}. \tag{5}$$

Indeed, (5) holds for atomic formulas by condition (4) and the assumption that $\alpha(x) \in T_0 \cup O$. Moreover, this property is preserved by propositional combinations, so it holds for all boolean combinations. Given an EBU sentence φ, we have:

Claim 1 *For each quantified subformula F of φ, for all valuations $\alpha : \mathsf{Vars} \to T_0 \cup O$, if $[\![F]\!]^{M_0,\alpha}$ then $[\![F]\!]^{M_T,\alpha}$.*

The base case corresponds to the previously proved case of quantifier-free formulas. We show that the condition is preserved under existential quantifiers, bounded universal quantifiers, and quantifiers over the finite set O. So suppose that $[\![F]\!]^{M_0,\alpha}$ implies $[\![F]\!]^{M_T,\alpha}$ for all $\alpha : \mathsf{Vars} \to T_0 \cup O$ and suppose that $\alpha : \mathsf{Vars} \to T_0 \cup O$ and $[\![F_1]\!]^{M_0,\alpha}$.

- Let $F_1 \equiv \exists v_t :: \mathsf{Tree}.\, F$. Then there exists $t \in T_0$ such that $[\![F]\!]^{M_0,\alpha'}$ where $\alpha = \alpha'[v_t := t]$. By induction hypothesis $[\![F]\!]^{M_T,\alpha'}$, so $[\![F_1]\!]^{M_T,\alpha}$.
- Let $F_1 \equiv \forall_S\, v_t :: \mathsf{Tree}.\, F$ for some set expression S. Then $[\![F]\!]^{M_0,\alpha[v_t := t]}$ for each $t \in [\![S]\!]^{M_0,\alpha}$. From (3), (4), $\alpha : \mathsf{Vars} \to T_0 \cup O$, and the fact that M_0 is subterm-closed, we conclude $[\![S]\!]^{M_T,\alpha} = [\![S]\!]^{M_0,\alpha} \subseteq T_0$. Consider arbitrary $t \in [\![S]\!]^{M_T,\alpha}$. Then $t \in [\![S]\!]^{M_0,\alpha}$, so $[\![F]\!]^{M_0,\alpha[v_t := t]}$. Because $t \in T_0$, by induction hypothesis $[\![F]\!]^{M_T,\alpha[v_t := t]}$. This proves $[\![F_1]\!]^{M_T,\alpha}$.
- The cases $F_1 \equiv \exists v_o :: \mathsf{Object}.\, F$ and $F_1 \equiv \forall v_o :: \mathsf{Object}.\, F$ are straightforward because the quantifiers are monotonic and the structures M_T and M_0 have the same domain of uninterpreted objects O.

This completes the proof of the claim, and the proof of ($\Longleftarrow$) direction of our statement. Note that we have not relied on the fact that M_T is full term model. In fact, this direction still holds for M_0 and M_1 where M_0 is a substructure of M_1 and M_1 is a substructure of M_T: if the EBU sentence holds in M_0, then it also holds in the larger substructure M_1. We will use this generalization in the proof of the converse direction.

Completeness ($\Longrightarrow$): Let φ be EBU sentence. We prove by induction that for all subformulas F of φ the following:

Claim 2 *For each $\alpha : \mathsf{Vars} \to T_T$, if $[\![F]\!]^{M_T,\alpha}$, then there exists a finite subterm-closed model M_0 and a valuation α_0 of M_0 such that $\alpha_0(x_i) = \alpha(x_i)$ for each variable x_i free in F, and such that $[\![F]\!]^{M_0,\alpha_0}$.*

The proof of this claim is by induction on the number of quantifiers in F.

For the base case, assume that F is quantifier-free, and let $x_1, \ldots, x_n$ be the variables of F. Then let $T_0 = [\![\{x_1, \ldots, x_n\}.*(\mathsf{left} \cup \mathsf{right})]\!]^{M_T,\alpha}$ and let M_0 be the substructure of M_T induced by T_0. Let $\alpha_0(x_i) = \alpha(x_i)$ for $1 \le i \le n$ and let $\alpha_0(v) = \alpha(x_1)$ for $v \notin \{x_1, \ldots, x_n\}$. Then $\alpha_0 : \mathsf{Vars} \to T_0 \cup O$, so by (5) we have $[\![F]\!]^{M_0,\alpha_0}$; we have thus identified the desired M_0 and α_0.

For the inductive step, assume that claim holds for formula F, we prove that it holds for F_1 which is the result of quantifying F. Suppose that $[\![F_1]\!]^{M_T,\alpha}$ holds. We consider several cases.

- $F_1 \equiv \exists v_t :: \mathsf{Tree}.\, F$. Then there exists $t \in T_T$ such that $[\![F]\!]^{M_T,\alpha'}$ where $\alpha' = \alpha[v_t := t]$. By the induction hypothesis, there exists α_0 that agrees with α' on the free variables of F and a finite subterm-closed model M_0 such that $[\![F]\!]^{M_0,\alpha_0}$. This means that $[\![F_1]\!]^{M_0,\alpha_0}$, and α_0 certainly agrees with α on the variables free in F_1.
- $F_1 \equiv \forall_S\, v_t :: \mathsf{Tree}.\, F$. Let $\bar{S} = [\![S]\!]^{M_T,\alpha}$. Assume first $\bar{S} \neq \emptyset$. Then for each $t \in \bar{S}$, if $\alpha(t) = \alpha[v_t := t]$, then $[\![F]\!]^{M_T,\alpha(t)}$, so by induction hypothesis there exists a model $M_0(t) = (T_0(t), O, \iota_0(t))$ and a valuation $\alpha_0(t)$ such that $[\![F]\!]^{M_0(t),\alpha_0(t)}$, and $\alpha_0(t)$ agrees with $\alpha(t)$ on the free variables of F. Then let $T_0' = \bar{S} \cup \bigcup_{t \in \bar{S}} T_0(t)$. Let T_0 be the subterm-closure of T_0', given by $T_0 = T_0 \cup \{t \mid \exists t' \in T_0'.\ (t', t) \in [\![\mathsf{subterm}]\!]^{M_T,\alpha}\}$. The union T_0' is finite because $\bar{S}$ is finite, and each $T_0(t)$ is finite. Therefore, the subterm-closure T_0 is finite, so there exists a finite subterm-closed structure $M_0 = (T_0, O, \iota_0)$. By the generalized version of the ($\Longrightarrow$) direction, because $M_0(t)$ is a substructure of M_0, we have that $[\![F]\!]^{M_0,\alpha_0(t)}$ for each $t \in \bar{S}$. Because we have $[\![S]\!]^{M_0,\alpha} = \bar{S}$ we conclude $[\![F_1]\!]^{M_0,\alpha_0(t_1)}$ where $t_1 \in \bar{S}$ is arbitrary.

  Next, consider the special case $\bar{S} = \emptyset$. Let
$$T_0 = [\![\{x_1, \ldots, x_n\}.*(\mathsf{left} \cup \mathsf{right})]\!]^{M_T,\alpha},$$
  where $x_1, \ldots, x_n$ are the free variables of S, and consider the corresponding model $M_0 = (T_0, O, \iota_0)$. Then $[\![S]\!]^{M_0,\alpha} = \emptyset$, so $[\![F_1]\!]^{M_0,\alpha}$.
- $F_1 \equiv \exists v_o :: \mathsf{Object}.\, F$. This case is analogous to the case $F_1 \equiv \exists v_t :: \mathsf{Tree}.F$.
- $F_1 \equiv \forall v_o :: \mathsf{Object}.\, F$. This case is similar to the case $F_1 \equiv \forall_S v_t :: \mathsf{Tree}.F$, but slightly simpler.